Unifying Theories of Programming in Isabelle

Simon Foster and Jim Woodcock

Department of Computer Science
University of York
York YO10 5GH
Great Britain
{jim.woodcock,simon.foster}@york.ac.uk
www.cs.york.ac.uk

Abstract. This is a tutorial introduction to the two most basic theories in Hoare & He's Unifying Theories of Programming and their mechanisation in the Isabelle interactive theorem prover. We describe the theories of relations and of designs (pre-postcondition pairs), interspersed with their formalisation in Isabelle and example mechanised proofs.

Keywords: Unifying Theories of Programming (UTP), Denotational Semantics, Laws of Programming, Isabelle, Interactive Theorem Proving.

Dedication: *To Professor He Jifeng on the occasion of his 70th birthday.*

1 Preliminaries

Unifying Theories of Programming, originally the work of Hoare & He [15], is a long-term research agenda that can be summarised as follows. Researchers have proposed many different programming theories and practitioners have proposed many different pragmatic programming paradigms; how do we understand the relationship between them?

UTP can trace its origins back to the work on predicative programming, which was started by Hehner; see [12] for a summary. It gives three principal ways to study such relationships: (i) by computational paradigm; (ii) by level of abstraction; and (iii) by method of presentation.

In Section 2, we introduce the basic concepts of UTP: alphabets, signatures, and healthiness conditions, and in Section 3 we outline the idea of *theory mechanisation* in Isabelle/HOL. In Section 4, we go on to describe the alphabetised relational calculus, the formalism used to describe predicates in UTP theories. In Section 5, we introduce a basic nondeterministic programming language and its laws of programming. In Section 6, we complete the initial presentation of UTP by describing the organisaiton of UTP theories into complete lattices. Sections 7 and 8 show how Hoare logic and the weakest precondition calculus can be defined in UTP. Section 9 introduces the UTP theory of designs that capture the notion of total correctness using assumptions and commitments. The paper ends with a discussion of related work (Section 11) and some conclusions including directions for future work (Section 12).

Z. Liu, J. Woodcock, and H. Zhu (Eds.): Theories of Programming, LNCS 8050, pp. 109–155, 2013.

Computational Paradigms. UTP groups programming languages according to a classification by computational model; for example, structured, object-oriented, functional, or logical. The technique is to identify common concepts and deal separately with additions and variations. It uses two fundamental scientific principles: (i) simplicity of presentation and (ii) separation of concerns.

Abstraction. Orthogonal to this organisation by computational paradigm, languages could be categorised by their level of abstraction within a particular paradigm. For example, the lowest level of abstraction may be the platform-specific technology of an implementation. At the other end of the spectrum, there might be a very high-level description of overall requirements and how they are captured and analysed. In between, there will be descriptions of components and descriptions of how they will be organised into architectures. Each of these levels will have interfaces specified by contracts of some kind. UTP gives ways of mapping between these levels based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code.

Presentation. The third classification is by the method chosen to present a language definition. There are three widely used scientific methods:

1. *Denotational*, in which each syntactic phrase is given a single mathematical meaning, a specification is just a set of denotations, and refinement is a simple correctness criterion of inclusion: every program behaviour is also a specification behaviour.
2. *Algebraic*, where no direct meaning is given to the language, but instead equalities relate different programs with the same meaning.
3. *Operational* where programs are defined by how they execute on an idealised abstract mathematical machine, giving a useful guide for compilation, debugging, and testing.

As Hoare & He point out [15], a comprehensive account of a programming theory needs all three kinds of presentation, and the UTP technique allows us to study differences and mutual embeddings, and to derive each from the others by mathematical definition, calculation, and proof.

The UTP research agenda has as its ultimate goal to cover all the interesting paradigms of computing, including both declarative and procedural, hardware and software. It presents a theoretical foundation for understanding software and systems engineering, and has been already been exploited in areas such as hardware ([23,39]), hardware/software co-design ([6]) and component-based systems ([38]). But it also presents an opportunity when constructing new languages, especially ones with heterogeneous paradigms and techniques.

Having studied the variety of existing programming languages and identified the major components of programming languages and theories, we can select theories for new, perhaps special-purpose languages. The analogy here is of a theory supermarket, where you shop for exactly those features you need while being confident that the theories plug-and-play together nicely.

A key concept in UTP is the *design*: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. Great use of this construct is made in the semantics of the *Circus* family of languages [35,21], where reactive processes are given a precondition-postcondition semantics that is then useful in assertional reasoning about state-rich reactive behaviour. Parts of this introduction are adapted from [36].

2 Introduction to UTP

The book by Hoare & He [15] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics; algebraic and operational semantics can then be proved sound against this model. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [28,33] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The alphabet is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, a program with variables x, y, and z would contain these names in its alphabet. Theories for particular programming paradigms require the observation of extra information; some examples are: a flag that says whether the program has started (ok); the current time ($clock$); the number of available resources (res); a trace of the events in the life of the program (tr); a set of refused events (ref); or a flag that says whether the program is waiting for interaction with its environment ($wait$).

The signature gives the rules for the syntax for denoting objects of the theory.

Healthiness conditions identify properties that characterise the predicates of the theory. Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Healthiness conditions (Hoare & He)).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.

   $$C \mathrel{\widehat{=}} clock \leq clock'$$

   If we add C to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas

clock′ describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that

$$P \Rightarrow C$$

2. The variable *ok* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

$$P \;=\; (ok \Rightarrow P)$$

If the program has not started, its behaviour is not described. □

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent: $P = \phi(P)$; this equation characterises the healthiness condition.

For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge C$, and then the following function on predicates $and_C \mathrel{\widehat{=}} \lambda X \bullet P \wedge C$ is the required idempotent. □

Example 2 (Boyle's Law).

Alphabet. Consider a simple theory to model the behaviour of a gas with regard to varying temperature and pressure. The physical phenomenon of the behaviour of the gas is subject to Boyle's Law:

> For a fixed amount of an ideal gas kept at a fixed temperature k, p (pressure) and V (volume) are inversely proportional (while one doubles, the other halves).

The alphabet of our theory contains the three mathematical variables described in Boyle's Law: k, p, and V. The model's observations correspond to real-world observations in what we might term *the model-based agenda*: the variables k, p, and V are *shared* with the real world.

Signature. We now need to describe the syntax used to denote objects of the theory. There is a requirement that temperature remains constant, so, to use our model to simulate the effects of Boyle's law, we need two just operations: (i) change the pressure; and (ii) change the volume. This pair of operations form the *signature* of our theory.

Healthiness Conditions. We know the observations we can make of our theory and the two operations we can use to change these observations. We now need to define some *healthiness conditions* as a way of determining membership of the theory. We are interested only in gases that obey Boyle's law, which states that $p * V = k$ must be *invariant*. Healthiness conditions determine the correct states of the system, and here we need both static and dynamic invariants:

- The equation $p * V = k$ is a *static* invariant: it applies to a *state*.
- We also require k to be constant. If we start in the state (k, p, V), where $p * V = k$, then transit to the state (k', p', V'), where $p' * V' = k'$, then we must have that $k' = k$. This is a *dynamic* invariant: it applies to a *relation*.

Some healthiness conditions can be defined using functions. Suppose $\alpha(\phi) = \{p, V, k\}$; then define $\boldsymbol{B}(\phi) = (\exists\, k \bullet \phi) \wedge (k = p * V)$. Now, regardless of whether ϕ is healthy or not, $\boldsymbol{B}(\phi)$ certainly is. For example:

$$\phi = (p = 10) \wedge (V = 5) \wedge (k = 100)$$

$$\begin{aligned}\boldsymbol{B}(\phi) &= (\exists\, k \bullet \phi) \wedge (k = p * V)\\ &= (\exists\, k \bullet (p = 10) \wedge (V = 5) \wedge (k = 100)) \wedge (k = p * V)\\ &= (p = 10) \wedge (V = 5) \wedge (k = p * V)\\ &= (p = 10) \wedge (V = 5) \wedge (k = 50)\end{aligned}$$

Notice that $\boldsymbol{B}(\boldsymbol{B}(\phi)) = \boldsymbol{B}(\phi)$. This is known as *idempotence*: taking the medicine twice leaves you healthy, no more and no less so than taking the medicine only once. This give us a simple test for healthiness: ϕ is already healthy if applying $\boldsymbol{B}$ leaves it unchanged. That is, if it satisfies the equation $\phi = \boldsymbol{B}(\phi)$. In this sense, ϕ is a fixed point of the idempotent function $\boldsymbol{B}$.

Consider another observation, that the pressure is between 10 and 20Pa:

$$\psi = (p \in 10\,..\,20) \wedge (V = 5)$$

Notice the fact that $\phi \Rightarrow \psi$. Now, if we make both ϕ and ψ healthy, we discover another fact: $\boldsymbol{B}(\phi) \Rightarrow \boldsymbol{B}(\psi)$.

$$\boldsymbol{B}(\phi) = (p = 10) \wedge (V = 5) \wedge (k = 50)$$

$$\boldsymbol{B}(\psi) = (p \in 10\,..\,20) \wedge (V = 5) \wedge (p * V = k)$$

$$(p = 10) \wedge (V = 5) \wedge (k = 50) \Rightarrow (p \in 10\,..\,20) \wedge (V = 5) \wedge (p * V = k)$$

In fact, $\boldsymbol{B}$ is *monotonic* in the sense that

$$\forall\, \phi, \psi \bullet (\phi \Rightarrow \psi) \Rightarrow (\boldsymbol{B}(\phi) \Rightarrow \boldsymbol{B}(\psi))$$

The most useful healthiness conditions are *monotonic idempotent functions*, which leads to some very important mathematical properties concerning complete lattices and Galois connections. □

Relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [15]. Here, we present the general relational setting, and the transition to the theory of designs.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 9 restricts the general theory to designs. Next, in Section 10, we present an alternative characterisation of the theory of designs using healthiness conditions. Finally, we conclude with a summary and a brief account of related work.

3 Theory Mechanisation

We have mechanised UTP in the interactive theorem prover *Isabelle/HOL* [19]. This allows the laws of programming to be mechanically verified, and make them available for use in mechnical program derivation, verification and refinement.

Interactive theorem provers (ITPs) have been built as an aid to programmers who wish to prove properties of their programs, such as correctness or refinement. Core to ITPs are proof goals or obligations: ostensible properties which must be discharged by the user under a given set of assumptions. A proof of such a goal consists of a sequence of calculations which transform the assumptions into the goal. The commands used in this transformation are called *proof tactics*, which help the programmer with varying degress of automation.

For instance we may wish to prove the simple property $\exists\, x.x > 6$. In Isabelle we can formalise such a property and proof in the following manner:

theorem *greater-than-six*: $\exists\, x{::}nat.\ x > 6$
  apply (*rule-tac* $x{=}7$ **in** *exI*)
  apply (*simp*)
done

There are two steps to this simple proof. We first invoke a rule called *exI* which performs *existential introduction*: we explictly supply a value for x for which the property holds, in this case 7. This leaves us with the proof goal $7 > 6$, which can be dispatched by simple arithmetic properties, so we use Isabelle's built in simplifier tactic simp to finish the proof. Isabelle then gives the message *"No subgoals!"*, which means the proof is complete and we can type done. At this point the property *greater-than-six* is entered into the property database for us to use in future proofs.

Mechanised proofs greatly increase the confidence that a given property is true. If we try to prove something which is not correct, Isabelle will not let us. For instance we can try and prove that *all* numbers are greater than six:

theorem *all-greater-than-six*: $\forall\, x{::}nat.\ x > 6$
  apply (*rule-tac allI*)
  — no possible progress

We cannot make much progress with such a proof – there just isn't a tactic to perform this proof as it is incorrect. In fact Isabelle also contains a helpful *counterexample generator* called nitpick [3] which can be used to see if a property can be refuted.

theorem *all-greater-than-six*: $\forall\ x{::}nat.\ x > 6$
nitpick

When we run this command Isabelle returns *"Nitpick found a counterexample: $x = 6$"*, which clearly shows why this proof is impossible. We therefore terminate our proof attempt by typing oops. So Isabelle acts as a theoretician's conscience, requiring that properties be comprehensively discharged. Isabelle proofs are also *correct-by-construction*. All proofs in Isabelle are constructed with respect to a small number of axioms in the Isabelle core, even those originating from automated proof tactics. This means that proofs are not predicated on the correctness of the tools and tactics, but only on the correctness of the underlying axioms which makes Isabelle proofs trustworthy.

Such proofs can, however, be tedious for a theortician to construct manually and therefore Isabelle provides a number of *automated proof tactics* to aid in proof. For instance the *greater-than-six* theorem can be proved in one step by application of Isabelle's main automated proof method auto. The auto tactic performs introduction/elimination style classical deduction and simplification in an effort to prove a goal. The user can also extend auto by adding additional rules which it can make use of, increasing the scope of problems which it can deal with.

Additionally, a more recent development is the addition of the sledgehammer [4] tool. Sledgehammer makes use of external first-order *automated theorem provers*. An automated theorem prover (ATP) is a system which can provide solutions to a certain subclass of logical problems. Sledgehammer can make use of a large number of ATPs, such as E [26], Vampire [24], SPASS [32], Waldmeister [13] and Z3 [9]. During a proof the user can invoke sledgehammer which causes the current goal, along with relevant assumptions, to be submitted to the ATPs which attempt a proof. Upon success, a proof command is returned which the user can insert to complete the proof.

For instance, let's say we wish to prove that for any given number there is an even number greater than it. We can prove such a property by calling sledgehammer:

theorem *greater-than-y-even*: $\forall\ y{::}nat.\ \exists\ x > y.\ (x\ mod\ 2 = 0)$
sledgehammer

In this case, sledgehammer successfully returns with ostensible proofs from four of the ATPs. We can select one of these proofs to see if it works:

theorem *greater-than-y-even*: $\forall\ y{::}nat.\ \exists\ x > y.\ (x\ mod\ 2 = 0)$
by (*metis Suc-1 even-Suc even-nat-mod-two-eq-zero lessI less-SucI*
numeral-1-eq-Suc-0 numeral-One)

The proof command is inserted and successfully discharges the goal, using a total of 7 laws from Isabelle's standard libary. In keeping with with proofs being correct by construction, sledgehammer does not trust the external ATPs to return sound results, but rather uses them as oracles whose proof output must be *reconstructed* with respect to Isabelle's axioms using the internally verified

prover metis. So Isabelle is a highly principled theorem prover in which trust can be placed, but also in which a high degree of proof automation can be obtained.

Sledgehammer works particularly well when used in concert with Isabelle's natural language proof script language *Isar*. Isar allows proof to be written in a calculational style familiar to mathematicians. The majority of proofs in tutorial are written Isar, as exemplified in Section 4.

4 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus [28,33], except that it is untyped and somewhat simpler. An *alphabetised predicate* (conventionally written as $P, Q, \ldots,$ ***true***) is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables (x, y, z, ...) and dashed variables (x', a', ...); the former represent initial observations, and the latter, observations made at a later intermediate or final point. The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* ($b, c, d, \ldots,$ *true*) has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. Of course, if a variable is mentioned in the alphabet of both P and Q, then they are both constraining the same variable.

The alphabetised relational calculus has been mechanised in Isabelle/UTP. An implementation of any calculus in computer science must make decisions about how unspecified details are fleshed out. For Isabelle/UTP this includes concretising the notions of types and values within alphabetised predicates. Isabelle/UTP predicates are parametrically polymorphic in the type of value which variables possess, and the user can supply their own notion of value, with an associated type system and function library. For instance, we are developing a value model for the *VDM* and *CML* specification languages. These will allow users to construct and verify VDM and CML specifications and programs. Indeed it is our hope that any programming language with a well-specified notion of values and types can be reasoned about within the UTP.

An Isabelle/UTP value model consists of four things:

1. A type to represent *values* (α).
2. A type to represent *types* (τ).
3. A *typing relation* ($\_:\_ :: \alpha \Rightarrow \tau \Rightarrow \mathsf{bool}$), specifies well-typed values.
4. A *definedness predicate* ($\mathcal{D} :: \alpha \Rightarrow \mathsf{bool}$), specifies well-defined values.

Variables within Isabelle/UTP contain a type which specifies the sort of data the variable should point to. The typing relation therefore allows us to realise predicates which are well-typed. The definedness predicate is used to determine when a value has no meaning. For instance it should follow that $\mathcal{D}(x/0) = \mathsf{false}$, whilst $\mathcal{D}1 = \mathsf{true}$. A correct program should never yield undefined values, and this predicate allows us to specify when this is and isn't the case. We omit details of a specific model since this has no effect on the mechanisation of the laws of UTP.

Isabelle/UTP predicates are modelled as sets of *bindings*, where a binding is a mapping from variables to well-typed values. The bindings contained within a predicate are those bindings which make the predicate true. For instance the predicate $x > 5$ is represented by the binding set $\{(x \mapsto 6), (x \mapsto 7), (x \mapsto 8) \cdots\}$. Likewise the predicate true is simply the set of all possible bindings, and false is the empty set $\emptyset$. We then define all the usual operators of predicate calculus which are used in these notes, including $\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\exists$ and $\forall$, most of which map onto binding set operators. An Isabelle/UTP relation is simply a predicate consisting only of dashed and undashed variables.

Isabelle/UTP provides a collection of tactics for aiding in automating proof. The overall aim is to achieve a level of automation such that proof can be at the same level as the standard pen-and-paper proofs contained herein, or even entirely automated. The three main tactics we have developed are as follows:

- utp-pred-tac – predicate calculus reasoning
- utp-rel-tac – relational calculus reasoning
- utp-expr-tac – expression evaluation

These tactics perform *proof by interpretation*. Isabelle/HOL already has a mature library of laws for reasoning about predicates and binary relations. Thus our tactics are designed to make use of these laws by identifying suitable subcalculi within the UTP for which well-known proof procedures exist. The tactics each also have a version in which auto is called after interpretation, for instance utp-pred-auto-tac is simply utp-pred-tac followed by auto. These tactics allow us to easily establish the basic laws of the UTP predicate and relational calculi.

Example 3 (Selection of basic predicate and relational calculus laws).

theorem *AndP-assoc*: ‘$P \wedge (Q \wedge R)$‘ = ‘$(P \wedge Q) \wedge R$‘
by (*utp-pred-tac*)

theorem *AndP-comm*: ‘$P \wedge Q$‘ = ‘$Q \wedge P$‘
by (*utp-pred-auto-tac*)

theorem *AndP-OrP-distr*: ‘$(P \vee Q) \wedge R$‘ = ‘$(P \wedge R) \vee (Q \wedge R)$‘
by (*utp-pred-auto-tac*)

theorem *AndP-contra*: ‘$P \wedge \neg P$‘ = *false*
by (*utp-pred-tac*)

theorem *ImpliesP-export*: ‘$P \Rightarrow Q$‘ = ‘$P \Rightarrow P \wedge Q$‘
by (*utp-pred-tac*)

theorem *SubstP-IffP*: ‘$(P \Leftrightarrow Q)[v/x]$‘ = ‘$P[v/x] \Leftrightarrow Q[v/x]$‘
by (*utp-pred-tac*)

theorem *SemiR-assoc*: $P\ ;\ (Q\ ;\ R) = (P\ ;\ Q)\ ;\ R$
by (*utp-rel-auto-tac*)

theorem *SemiR-SkipR-right*: $P\ ;\ II = P$
by (*utp-rel-tac*)

Using the tactics we have constructed a large library of algebraic laws for propositional logic and relation algebra. These laws are most easily applied by application of sledgehammer, which will find the most appropriate rules to complete the step of proof. Sledgehammer works particularly well when used in concert with Isabelle's natural language proof script language *Isar*. Isar allows proof to be written in a calculational style familiar to mathematicians. We will cover these in detail in the next section.

5 Laws of Programming

A distinguishing feature of UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [15]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is given simply as $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \textbf{\textit{iff}} \quad [\,P \Rightarrow S\,]$$

Example 4 (Refinement). Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness is argued as follows.

$$\begin{aligned}
& x' > x \wedge y' = y \ \sqsubseteq\ x' = x + 1 \wedge y' = y && \textit{definition of } \sqsubseteq \\
&= [\,x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y\,] && \textit{universal one-point rule, twice} \\
&= [\,x + 1 > x \wedge y = y\,] && \textit{arithmetic and reflection} \\
&= \mathit{true}
\end{aligned}$$

And so, the refinement is valid. □

In the following sections, we introduce the definitions of the constructs of a nondeterministic sequential programming language, together with their laws of programming. Each law can be proved correct as a theorem involving the denotational semantics given by its definition. The constructs are: (i) conditional choice; (ii) sequential composition; (iii) assignment; (iv) nondeterminism; and (v) variable blocks.

5.1 Conditional

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q) \qquad \textbf{if } \alpha b \subseteq \alpha P = \alpha Q$$
$$\alpha(P \lhd b \rhd Q) \;\widehat{=}\; \alpha P$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

L1	$P \lhd b \rhd P = P$	*idempotence*
L2	$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	*symmetry*
L3	$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4	$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5	$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$	*unit*
L6	$P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable branch*
L7	$P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$	*disjunction*
L8	$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$	*interchange*
L9	$\neg\, (P \lhd b \rhd Q) = (\neg\, P \lhd b \rhd \neg\, Q)$	*cond. neg.*
L10	$(P \lhd b \rhd Q) \wedge \neg\, (R \lhd b \rhd S) = (P \wedge \neg\, R) \lhd b \rhd (Q \wedge \neg\, S)$	*comp.*
L11	$(P \Rightarrow (Q \lhd b \rhd R)) = ((P \Rightarrow Q) \lhd b \rhd (P \Rightarrow R))$	*cond.-⇒-1*
L12	$((P \lhd b \rhd Q) \Rightarrow R) = ((P \Rightarrow R) \lhd b \rhd (Q \Rightarrow R))$	*cond.-⇒-2*
L13	$(P \lhd b \rhd Q) \wedge R = (P \wedge R) \lhd b \rhd (Q \wedge R)$	*cond.-conjunction*
L14	$(P \lhd b \rhd Q) \vee R = (P \vee R) \lhd b \rhd (Q \vee R)$	*cond.-disjunction*
L15	$b \wedge (P \lhd b \rhd Q) = (b \wedge P)$	*cond.-left-simp*
L16	$\neg\, b \wedge (P \lhd b \rhd Q) = (\neg\, b \wedge Q)$	*cond.-right-simp*
L17	$(P \lhd b \rhd Q) = ((b \wedge P) \lhd b \rhd Q)$	*cond.-left*
L18	$(P \lhd b \rhd Q) = (P \lhd b \rhd (\neg\, b \wedge Q))$	*cond.-right*

In the Interchange Law (**L8**), the symbol $\odot$ stands for any truth-functional operator. For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition. As an example, we present the proof of the Unreachable Branch Law (**L6**).

*Example 5 (Proof of Unreachable Branch (**L6**)).*

$$\begin{aligned}
&(P \lhd b \rhd (Q \lhd b \rhd R)) && \textbf{L2}\\
&= ((Q \lhd b \rhd R) \lhd \neg\, b \rhd P) && \textbf{L3}\\
&= (Q \lhd b \wedge \neg\, b \rhd (R \lhd \neg\, b \rhd P)) && \textit{propositional calculus}
\end{aligned}$$

$= (Q \lhd \mathit{false} \rhd (R \lhd \neg\ b \rhd P))$ **L5**

$= (R \lhd \neg\ b \rhd P)$ **L2**

$= (P \lhd b \rhd R)$ □

This proof can be mechanised in Isar using the same sequence:

*Example 6 (Isar Proof of Unreachable Branch (**L6**)).*

```
theorem CondR-unreachable-branch:
  ‘(P ◁ b ▷ (Q ◁ b ▷ R))‘ = ‘P ◁ b ▷ R‘ (is ?lhs = ?rhs)
proof −
  have ?lhs     = ‘((Q ◁ b ▷ R) ◁ ¬b ▷ P)‘ by (metis CondR-sym)
  also have ... = ‘(Q ◁ b ∧ ¬ b ▷ (R ◁ ¬ b ▷ P))‘ by (metis CondR-assoc)
  also have ... = ‘(Q ◁ false ▷ (R ◁ ¬ b ▷ P))‘ by (utp-pred-tac)
  also have ... = ‘(R ◁ ¬ b ▷ P)‘ by (metis CondR-false)
  also have ... = ?rhs by (metis CondR-sym)
  finally show ?thesis .
qed
```

Isar provides an environment to perform proofs in a natural style, by proving subgoals and the combining these to produce the final goal. The **proof** command opens an Isar proof environment for a goal, and **have** is used to create a subgoal to act as lemma for the overall goal, which must be followed by a proof, usually added using the **by** command. In a calculational proof, we want to transitively compose the previous subgoal with the next, which is done by prefixing the subgoal with **also**. Furthermore Isar provides the ... variable which contains the right-hand side of the previous subgoal. Once all steps of the proof are complete the **finally** command collects all the subgoals together, and **show** is used to prove the overall goal. In the case that no further proof is needed the user can simply type **.** to finish. A completed proof environment can then be terminate with **qed**.

In this case, the proof proceeds by application of sledgehammer for each line where an algebraic law is applied, and by utp-pred-tac when propositional calculus is needed.

Implication is, of course, still the basis for reasoning about the correctness of conditionals. We can, however, prove refinement laws that support a compositional reasoning technique.

Law 51 (Refinement to conditional)

$$P \sqsubseteq (Q \lhd b \rhd R) \ = \ (\, P \sqsubseteq b \wedge Q\,) \wedge (\, P \sqsubseteq \neg\ b \wedge R\,) \qquad \square$$

This result allows us to prove the correctness of a conditional by a case analysis on the correctness of each branch. Its proof is as follows.

Proof of Law 51

$P \ \sqsubseteq \ (Q \lhd b \rhd R)$ — *definition of* $\sqsubseteq$

$= [\,(Q \lhd b \rhd R) \Rightarrow P\,]$ — *definition of conditional*

$= [\, b \wedge Q \vee \neg\, b \wedge R \Rrightarrow P \,]$ *propositional calculus*
$= [\, b \wedge Q \Rrightarrow P \,] \wedge [\, \neg\, b \wedge R \Rrightarrow P \,]$ *definition of $\sqsubseteq$, twice*
$= (\, P \sqsubseteq b \wedge Q \,) \wedge (\, P \sqsubseteq \neg\, b \wedge R \,)$ □

The corresponding proof in Isar can also be given:

Example 7 (Isar Proof of Law 51).

theorem *RefineP-to-CondR*:
‘$P \sqsubseteq (Q \lhd b \rhd R)$‘ = ‘$(P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg\, b \wedge R)$‘
proof −
have ‘$P \sqsubseteq (Q \lhd b \rhd R)$‘ = ‘$[(Q \lhd b \rhd R) \Rrightarrow P]$‘ **by** (*metis RefP-def*)
also have ... = ‘$[(b \wedge Q) \vee (\neg\, b \wedge R) \Rrightarrow P]$‘ **by** (*metis CondR-def*)
also have ... = ‘$[b \wedge Q \Rrightarrow P] \wedge [\neg\, b \wedge R \Rrightarrow P]$‘ **by** (*utp-pred-auto-tac*)
also have ... = ‘$(P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg\, b \wedge R)$‘ **by** (*metis RefP-def*)
finally show *?thesis* .
qed

A compositional argument is also available for conjunctions.

Law 52 (Separation of requirements)

$$((P \wedge Q) \sqsubseteq R) = (P \sqsubseteq R) \wedge (Q \sqsubseteq R)$$ □

We can prove that an implementation satisfies a conjunction of requirements by considering each conjunct separately. The omitted proof is left as an exercise for the interested reader.

5.2 Sequential Composition

Sequence is modelled as relational composition. Two relations may be composed, providing that the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$P(v') \,;\, Q(v) \;\widehat{=}\; \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textbf{\textit{if}}\ out\alpha P = in\alpha Q' = \{v'\}$$
$$in\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; in\alpha P$$
$$out\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; out\alpha Q$$

Composition is associative and distributes backwards through the conditional.

L1 $P \,;\, (Q \,;\, R) = (P \,;\, Q) \,;\, R$ *associativity*
L2 $(P \lhd b \rhd Q) \,;\, R = ((P \,;\, R) \lhd b \rhd (Q \,;\, R))$ *left distribution*

The simple proofs of these laws, and those of a few others in the sequel, are omitted for the sake of conciseness.

5.3 Assignment

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

$$x :=_A e \ \widehat{=}\ (x' = e \land y' = y \land \cdots \land z' = z)$$
$$\alpha(x :=_A e) \ \widehat{=}\ A \cup A'$$

There is a degenerate form of assignment that changes no variable: it is called "skip", and has the following definition.

$$\Pi_A \ \widehat{=}\ (v' = v) \qquad \textbf{if } A = \{v\}$$
$$\alpha\Pi_A \ \widehat{=}\ A \cup A'$$

L1 $(x := e) = (x, y := e, y)$ *framing*

L2 $(x, y, z := e, f, g) = (y, x, z := f, e, g)$ *permutation*

L3 $(x := e \ ; x := f(x)) = (x := f(e))$ *composition*

L4 $(x := e \ ; (P \lhd b(x) \rhd Q)) = ((x := e \ ; P) \lhd b(e) \rhd (x := e \ ; Q))$

L5 $P \ ; \Pi_{\alpha P} = P = \Pi_{\alpha P} \ ; P$ *unit*

L6 $v' = e \ ; P \ = \ P[e/v]$ where $\alpha P = \{v, v'\}$ *left-one-point*

L7 $P \ ; v = e \ = \ P[e/v']$ where $\alpha P = \{v, v'\}$ *right-one-point*

5.4 Nondeterminism

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$P \sqcap Q \ \widehat{=}\ P \lor Q \qquad \textbf{if } \alpha P = \alpha Q$$
$$\alpha(P \sqcap Q) \ \widehat{=}\ \alpha P$$

The alphabet must be the same for both arguments.

L1 $P \sqcap Q = Q \sqcap P$ *symmetry*

L2 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ *associativity*

L3 $P \sqcap P = P$ *idempotence*

L4 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$ *distrib.*

L5 $(P \lhd b \rhd (Q \sqcap R)) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$ *distrib.*

L6 $(P \sqcap Q) \ ; R = (P \ ; R) \sqcap (Q \ ; R)$ *distrib.*

L7 $P\ ;\ (Q \sqcap R) = (P\ ;\ Q) \sqcap (P\ ;\ R)$ *distrib.*

L8 $P \sqcap (Q \lhd b \rhd R) = ((P \sqcap Q) \lhd b \rhd (P \sqcap R))$ *distrib.*

The following law gives an important property of refinement: if P is refined by Q, then offering the choice between P and Q is immaterial; conversely, if the choice between P and Q behaves exactly like P, so that the extra possibility of choosing Q does not add any extra behaviour, then Q is a refinement of P.

Law 53 (Refinement and nondeterminism)

$$P \sqsubseteq Q \ = \ (P \sqcap Q = P) \qquad \square$$

Proof

$$\begin{array}{ll} & P \sqcap Q = P & \textit{antisymmetry} \\ = & (P \sqcap Q \sqsubseteq P) \wedge (P \sqsubseteq P \sqcap Q) & \textit{definition of } \sqsubseteq\textit{, twice} \\ = & [\, P \Rightarrow P \sqcap Q \,] \wedge [\, P \sqcap Q \Rightarrow P \,] & \textit{definition of } \sqcap\textit{, twice} \\ = & [\, P \Rightarrow P \vee Q \,] \wedge [\, P \vee Q \Rightarrow P \,] & \textit{propositional calculus} \\ = & \mathit{true} \wedge [\, P \vee Q \Rightarrow P \,] & \textit{propositional calculus} \\ = & [\, Q \Rightarrow P \,] & \textit{definition of } \sqsubseteq \\ = & P \sqsubseteq Q & \square \end{array}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 54 (Thin nondeterminism)

$$P \sqcap Q \sqsubseteq P \qquad \square$$

The proof is immediate from properties of the propositional calculus.

5.5 Alphabet Extension

Alphabet extension is a way adding new variables to the alphabet of a predicate, for example, when new programming variables are declared, as we see in the next section.

$$R_{+x} \ \widehat{=}\ R \wedge x' = x \qquad \text{for} \quad x, x' \notin \alpha R$$
$$\alpha(R_{+x}) \ \widehat{=}\ \alpha R \cup \{x, x'\}$$

5.6 Variable Blocks

Variable blocks are split into the commands ***var*** x, which declares and introduces x in scope, and ***end*** x, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\textbf{\textit{var}}\ x \ \widehat{=}\ (\exists\, x \bullet \mathit{II}_A) \qquad \alpha(\ \textbf{\textit{var}}\ x\ ) \ \widehat{=}\ A \setminus \{x\}$$
$$\textbf{\textit{end}}\ x \ \widehat{=}\ (\exists\, x' \bullet \mathit{II}_A) \qquad \alpha(\ \textbf{\textit{end}}\ x\ ) \ \widehat{=}\ A \setminus \{x'\}$$

The relation ***var*** x is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, ***end*** x includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$(\ \textbf{var}\ x\ ;\ Q\ ) = (\ \exists\, x \bullet Q\ )$$
$$(\ Q\ ;\ \textbf{end}\ x\ ) = (\ \exists\, x' \bullet Q\ )$$

More interestingly, we can use $\textbf{var}\ x$ and $\textbf{end}\ x$ to specify a variable block.

$$(\ \textbf{var}\ x\ ;\ Q\ ;\ \textbf{end}\ x\ ) = (\ \exists\, x, x' \bullet Q\ )$$

In programs, we use $\textbf{var}\ x$ and $\textbf{end}\ x$ paired in this way, but the separation is useful for reasoning.

The following laws are representative.

L6 $(\ \textbf{var}\ x\ ;\ \textbf{end}\ x\ ) = \mathit{II}$

L8 $(\ x := e\ ;\ \textbf{end}\ x\ ) = (\ \textbf{end}\ x\ )$

Variable blocks introduce the possibility of writing programs and equations like that below.

$$(\ \textbf{var}\ x\ ;\ x := 2 * y\ ;\ w := 0\ ;\ \textbf{end}\ x\ )$$
$$=\ (\ \textbf{var}\ x\ ;\ x := 2 * y\ ;\ \textbf{end}\ x\ )\ ;\ w := 0$$

Clearly, the assignment to w may be moved out of the scope of the the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and suppose that $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension*. If the right-hand assignment is $P \mathrel{\widehat{=}} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

$$P_{+x} \mathrel{\widehat{=}} P \wedge x' = x \qquad \text{for}\ \ x, x' \notin \alpha P$$
$$\alpha(P_{+x}) \mathrel{\widehat{=}} \alpha P \cup \{x, x'\}$$

If Q does not mention x, then the following laws hold.

L1 $\textbf{var}\ x\ ;\ Q_{+x}\ ;\ P\ ;\ \textbf{end}\ x\ =\ Q\ ;\ \textbf{var}\ x\ ;\ P\ ;\ \textbf{end}\ x$
L2 $\textbf{var}\ x\ ;\ P\ ;\ Q_{+x}\ ;\ \textbf{end}\ x\ =\ \textbf{var}\ x\ ;\ P\ ;\ \textbf{end}\ x\ ;\ Q$

Together with the laws for variable declaration and undeclaration, the laws of alphabet extension allow for program transformations that introduce new variables and assignments to them.

6 The Complete Lattice

A lattice is a partially ordered set where all non-empty finite subsets have both a least upper-bound (join) and a greatest lower-bound (meet). A complete lattice additionally requires all subsets have both a join and a meet.

Example 8 (Complete lattice: Powerset). The powerset of any set S, ordered by inclusion, forms a complete lattice. The empty set is the least element and S itself is the greatest element. Set union is the join operation and set intersection is the meet. For example, the powerset of $\{0, 1, 2, 3\}$ ordered by subset, is illustrated in Figure 1. □

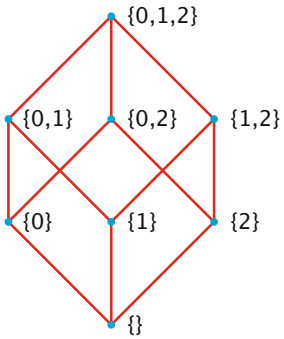

Fig. 1. $0 \,..\, 3$ ordered by inclusion

Example 9 (Complete lattice: Divisible natural numbers). Natural numbers ordered by divisibility form a complete lattice. Natural number n is exactly divisible by another natural number m, providing n is an exact multiple of m. This gives us the following partial order:

$$m \sqsubseteq n \Leftrightarrow (\exists\, k \bullet k \times m = n)$$

In this ordering, 1 is the bottom element (it exactly divides every other number) and 0 is the top element (it can be divided exactly by every other number). For example, if we restrict our attention to the numbers between 0 and 1, we obtain the lattice illustrated in Figure 2. □

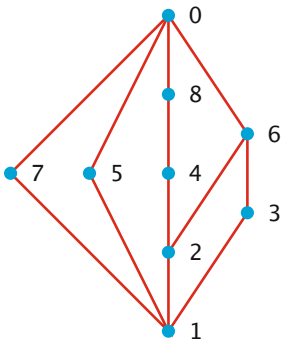

Fig. 2. $0 \,..\, 8$ ordered by divisibility

Isabelle/HOL contains a comprehensive mechanised theory of complete lattices and fixed-points, which we directly make use of in Isabelle/UTP. We therefore omit details of these proofs' mechanisation; the reader can refer directly to the HOL library.

6.1 Lattice Operators

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_A$, and is the weakest predicate ***true***; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

L1 $\bot_A \sqsubseteq P$ *bottom element*

L2 $P \sqsubseteq \top_A$ *top element*

The least upper bound is not defined in terms of the relational model, but by the law ***L1*** below. This law alone is enough to prove laws ***L1A*** and ***L1B***, which are actually more useful in proofs.

L1 $P \sqsubseteq (\bigsqcap S)$ ***iff*** $(P \sqsubseteq X$ ***for all*** X ***in*** $S)$ *unbounded nondeterminism*

L1A $(\bigsqcap S) \sqsubseteq X$ ***for all*** X ***in*** S *lower bound*

L1B ***if*** $P \sqsubseteq X$ ***for all*** X ***in*** S, ***then*** $P \sqsubseteq (\bigsqcap S)$ *greatest lower bound*

L2 $(\bigsqcup S) \sqcap Q = \bigsqcup \{ P \sqcap Q \mid P \in S \}$

L3 $(\bigsqcap S) \sqcup Q = \bigsqcap \{ P \sqcup Q \mid P \in S \}$

L4 $(\bigsqcap S) \mathbin{;} Q = \bigsqcap \{ P \mathbin{;} Q \mid P \in S \}$

L5 $R \mathbin{;} (\bigsqcap S) = \bigsqcap \{ R \mathbin{;} P \mid P \in S \}$

These laws characterise basic properties of least upper bounds.

As we saw above, a function F is *monotonic* if and only if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation and conjunction are not. There is a class of operators that are all monotonic.

Example 10 (Disjunctivity and monotonicity). Suppose that $P \sqsubseteq Q$ and that $\odot$ is disjunctive, or rather, $R \odot (S \sqcap T) = (R \odot S) \sqcap (R \odot T)$. From this, we can conclude that $P \odot R$ is monotonic in its first argument.

$$
\begin{aligned}
& P \odot R && \textit{assumption } (P \sqsubseteq Q) \textit{ and Law 53} \\
&= (P \sqcap Q) \odot R && \textit{assumption } (\odot \textit{ disjunctive}) \\
&= (P \odot R) \sqcap (Q \odot R) && \textit{thin nondeterminism} \\
&\sqsubseteq Q \odot R
\end{aligned}
$$

A symmetric argument shows that $P \odot Q$ is also monotonic in its other argument. In summary, disjunctive operators are always monotonic. The converse is not true: monotonic operators are not always disjunctive. □

6.2 Recursion

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed-point. Even more, a result by Knaster and Tarski, described below, says that the set of fixed-points form a complete lattice themselves. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed-point of $X = P \fatsemi X$, and $\bot$ is the weakest.

Example 11 (Complete lattice of fixed points). Consider the function $f(s) = s \cup \{0\}$ restricted to the domain comprising the powerset of $\{0, 1, 2\}$. The complete lattice of fixed points for f is illustrated in Fig 3. □

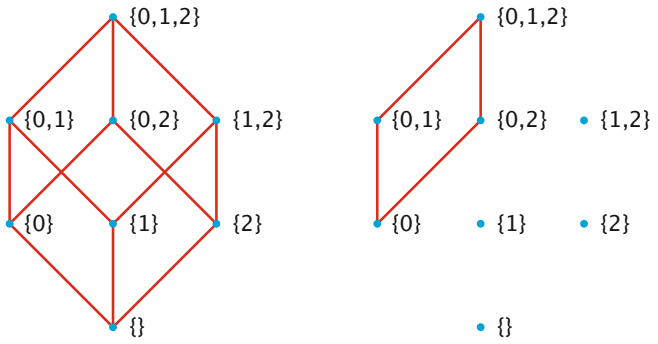

Fig. 3. Complete lattice of fixed points of the function $f(s) = s \cup \{0\}$ (right)

Let $\sqsubseteq$ be a partial order in a lattice X and let $F : X \to X$ be a function over X. A pre-fixed-point of F is any x such that $F(x) \sqsubseteq x$, and a post-fixed-point of f is any x such that $x \sqsubseteq F(x)$. The Knaster-Tarski theorem states that a monotonic function F on a complete lattice has three properties: (i) The function F has at least one fixed point. (ii) The weakest fixed-point of F coincides with the greatest lower-bound of the set of its post-fixed-points; similarly, the strongest fixed-point coincides with the least upper-bound of the set of its pre-fixed-points. (iii) The set of fixed points of F form a complete lattice.

The weakest fixed-point of the function F is denoted by $\mu\, F$, and is defined simply the greatest lower bound (the *weakest*) of all the pre-fixed-points of F.

$$\mu\, F \mathrel{\widehat{=}} \bigsqcap \{\, X \mid F(X) \sqsubseteq X \,\}$$

The strongest fixed-point νF is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion, where they write a recursive program as $\mu\, X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, called the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu\, X \bullet \mathcal{C}(X) \mathrel{\widehat{=}} \mu\, F \qquad \textbf{\textit{where}}\ F \mathrel{\widehat{=}} \lambda\, X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed-points are valid.

L1	$\mu\, F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$	*weakest fixed-point*
L2	$F(\mu\, F) = \mu\, F$	*fixed-point*

L1 establishes that $\mu\, F$ is weaker than any fixed-point; ***L2*** states that $\mu\, F$ is itself a fixed-point. From a programming point of view, ***L2*** is just the copy rule.

Proof of ***L1***

$$\begin{array}{lr}
\quad F(Y) \sqsubseteq Y & \textit{set comprehension} \\
= Y \in \{\, X \mid F(X) \sqsubseteq X \,\} & \textit{lattice law } \textbf{\textit{L1A}} \\
\Rightarrow \bigsqcap \{\, X \mid F(X) \sqsubseteq X \,\} \sqsubseteq Y & \textit{definition of } \mu\, F \\
= \mu\, F \sqsubseteq Y & \Box
\end{array}$$

Proof of ***L2***

$$\begin{array}{lr}
\quad \mu\, F = F(\mu\, F) & \textit{mutual refinement} \\
= \mu\, F \sqsubseteq F(\mu\, F) \wedge F(\mu\, F) \sqsubseteq \mu\, F & \textit{fixed-point law } \textbf{\textit{L1}} \\
\Leftarrow F(F(\mu\, F)) \sqsubseteq F(\mu\, F) \wedge F(\mu\, F) \sqsubseteq \mu\, F & F \textit{ monotonic} \\
\Leftarrow F(\mu\, F) \sqsubseteq \mu\, F & \textit{definition} \\
= F(\mu\, F) \sqsubseteq \bigsqcap \{\, X \mid F(X) \sqsubseteq X \,\} & \textit{lattice law } \textbf{\textit{L1B}} \\
\Leftarrow \forall\, X \in \{\, X \mid F(X) \sqsubseteq X \,\} \bullet F(\mu\, f) \sqsubseteq X & \textit{comprehension} \\
= \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq X & \textit{transitivity of } \sqsubseteq \\
\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq F(X) & F \textit{ monotonic} \\
\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow \mu\, F \sqsubseteq X & \textit{fixed-point law } \textbf{\textit{L1}} \\
= \mathit{true} & \Box
\end{array}$$

6.3 Iteration

The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed-point of a conditional expression.

$$b * P \;\widehat{=}\; \mu\, X \bullet ((P \,;\, X) \lhd b \rhd \mathit{II})$$

Example 12 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $\mathit{true} * \mathit{II}$, which has the semantics $\mu\, X \bullet X$.

$$\begin{array}{ll} \quad \mu\, X \bullet X & \textit{definition of least fixed-point} \\ = \bigsqcap \{\, Y \mid (\lambda\, X \bullet X)(Y) \sqsubseteq Y \,\} & \textit{function application} \\ = \bigsqcap \{\, Y \mid Y \sqsubseteq Y \,\} & \textit{reflexivity of } \sqsubseteq \\ = \bigsqcap \{\, Y \mid \mathit{true} \,\} & \textit{property of } \bigsqcap \\ = \bot & \square \end{array}$$

$\square$

A surprising, but simple, consequence of Example 12 is that a program can recover from a non-terminating loop!

Example 13 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$\begin{array}{ll} \quad (b * P);\; x := c & \textit{Example 12} \\ = \bot;\; x := c & \textit{definition of } \bot \\ = \textbf{\textit{true}};\; x := c & \textit{definition of assignment} \\ = \textbf{\textit{true}};\; x' = c & \textit{definition of composition} \\ = \exists\, x_0 \bullet \textbf{\textit{true}} \wedge x' = c & \textit{predicate calculus} \\ = x' = c & \textit{definition of assignment} \\ = x := c & \square \end{array}$$

$\square$

Example 13 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model; we return to this in Section 9.

7 Hoare Logic

The Hoare triple $p \left\{ Q \right\} r$ is a specification of the correctness of a program Q. Here, p and r are assertions and Q is a command. This is partial correctness

in the sense that the assertions do not require Q to terminate. Instead, the correctness statement is that, if Q is started in a state satisfying p, then, if it does terminate, it will finish in a state satisfying r. We define the meaning of the Hoare triple as a universal implication:

$$p \{ Q \} r \mathrel{\widehat{=}} [p \wedge Q \Rightarrow r']$$

This is a correctness assertion that can be expressed as the refinement assertion $(p \Rightarrow r') \sqsubseteq Q$.

The laws that can be proved from this definition form the Axioms of Hoare Logic:

L1 **if** $p \{ Q \} r$ **and** $p \{ Q \} s$ **then** $p \{ Q \} (r \wedge s)$

L2 **if** $p \{ Q \} r$ **and** $q \{ Q \} r$ **then** $(p \vee q) \{ Q \} r$

L3 **if** $p \{ Q \} r$ **then** $(p \wedge q) \{ Q \} (r \vee s)$

L4 $r(e) \{ x := e \} r(x)$

L5 **if** $(p \wedge b) \{ Q_1 \} r$ **and** $(p \wedge \neg b) \{ Q_2 \} r$ **then**
$p \{ Q_1 \lhd b \rhd Q_2 \} r$

L6 **if** $p \{ Q_1 \} s$ **and** $s \{ Q_2 \} r$ **then** $p \{ Q_1 \,;\, Q_2 \} r$

L7 **if** $p \{ Q_1 \} r$ **and** $p \{ Q_2 \} r$ **then** $p \{ Q_1 \sqcap Q_2 \} r$

L8 **if** $(b \wedge c) \{ Q \} c$ **then** $c \{ \nu X \bullet (Q \,;\, X) \lhd b \rhd \mathit{II} \} (\neg\, b \wedge c)$

L9 $\mathit{false} \{ Q \} r$ **and** $p \{ Q \} \mathit{true}$ **and** $p \{ \mathbf{false} \} \mathit{false}$ **and** $p \{ \mathit{II} \} p$

Proof of **L1**.

$$\begin{array}{ll}
& (p \{ Q \} r) \wedge (p \{ Q \} s) \\
= & (Q \Rightarrow (p \Rightarrow r')) \wedge (Q \Rightarrow (p \Rightarrow s')) \\
= & (Q \Rightarrow (p \Rightarrow r') \wedge (p \Rightarrow s')) \\
= & (Q \Rightarrow (p \Rightarrow r' \wedge s')) \\
= & p \{ Q \} (r \wedge s) & [\Box]
\end{array}$$

Proof of **L8**: Suppose that $(b \wedge c) \{ Q \} c$. Define $Y \mathrel{\widehat{=}} c \Rightarrow \neg\, b' \wedge c'$

$$\begin{array}{lr}
& c \{ \nu X \bullet (Q \,;\, X) \lhd b \rhd \mathit{II} \} (\neg\, b \wedge c) \\
= & Y \sqsubseteq \nu X \bullet (Q \,;\, X) \lhd b \rhd \mathit{II} & \textit{by definition} \\
\Leftarrow & Y \sqsubseteq (Q \,;\, Y) \lhd b \rhd \mathit{II} & \textit{by sfp L1} \\
= & (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge (\, Y \sqsubseteq \neg\, b \wedge \mathit{II} \,) & \textit{refinement to cond} \\
= & (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge [\neg\, b \wedge \mathit{II} \Rightarrow (c \Rightarrow \neg\, b' \wedge c')] & \textit{by def} \\
= & (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge \mathit{true} & \text{predicate calculus} \\
= & c \{ b \wedge Q \,;\, Y \} (\neg\, b \wedge c) & \textit{by definition} \\
\Leftarrow & (\, c \{ b \wedge Q \} c \,) \wedge (\, c \{ c \Rightarrow \neg\, b' \wedge c' \} \neg\, b \wedge c \,) & \textit{by Hoare L6} \\
= & \mathit{true} & \textit{by assumption and predicate calculus}
\end{array}$$

$\Box$

8 Weakest Preconditions

A Hoare triple involves three variables: a precondition, a postcondition, and a command. If we fix two of these variables, then we can calculate an extreme solution for the third. For example, if we fix the command and the precondition, then we calculate the strongest postcondition. Alternatively, we could fix the command and the postcondition and calculate the weakest precondition, and that is what we do here. We start with some relational calculus to obtain an implication with the precondition assertion as the antecedent of an implication of the form: $[\, p \Rightarrow R \,]$. If we fix R, then there are perhaps many solutions for p that satisfy this inequality. Of all the possibilities, the weakest must actually be equal to R.

$$
\begin{aligned}
& p(v) \left\{ \, Q(v, v') \, \right\} r(v) \\
& = [\, Q(v, v') \Rightarrow (\, p(v) \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow (\, Q(v, v') \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow (\forall\, v' \bullet Q(v, v') \Rightarrow r(v') \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\exists\, v' \bullet Q(v, v') \wedge \neg\, r(v') \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\exists\, v_0 \bullet Q(v, v_0) \wedge \neg\, r(v_0) \,) \,] \\
& = [\, p(v) \Rightarrow \neg\, (\, Q(v, v') \,;\, \neg\, r(v) \,) \,]
\end{aligned}
$$

So now, if we take $\mathcal{W}(v) = \neg\, (\, Q(v, v') \,;\, \neg\, r(v) \,)$, then the following Hoare triple must be valid:

$$\mathcal{W}(v) \left\{ \, Q(v, v') \, \right\} r(v)$$

Here, $\mathcal{W}$ is the weakest solution for the precondition for Q to be guaranteed to achieve r.

We define the predicate transformer ***wp*** as a relation between Q and r as follows:

$$Q \mathbin{\textbf{\textit{wp}}} r \;\widehat{=}\; \neg\, (Q \,;\, \neg\, r)$$

The laws for the weakest precondition operator are as follows:

L1 $((x := e) \mathbin{\textbf{\textit{wp}}} r(x)) = r(e)$

L2 $((P \,;\, Q) \mathbin{\textbf{\textit{wp}}} r) = (P \mathbin{\textbf{\textit{wp}}} (Q \mathbin{\textbf{\textit{wp}}} r))$

L3 $((P \lhd b \rhd Q) \mathbin{\textbf{\textit{wp}}} r) = ((P \mathbin{\textbf{\textit{wp}}} r) \lhd b \rhd (Q \mathbin{\textbf{\textit{wp}}} r))$

L4 $((P \sqcap Q) \mathbin{\textbf{\textit{wp}}} r) = (P \mathbin{\textbf{\textit{wp}}} r) \wedge (Q \mathbin{\textbf{\textit{wp}}} r)$

L5 if $[\, r \Rightarrow s \,]$ then $[\, (Q \mathbin{\textbf{\textit{wp}}} r) \Rightarrow (Q \mathbin{\textbf{\textit{wp}}} s) \,]$

L6 if $[\, Q \Rightarrow S \,]$ then $[\, (S \mathbin{\textbf{\textit{wp}}} r) \Rightarrow (Q \mathbin{\textbf{\textit{wp}}} r) \,]$

L7 $(Q \mathbin{\textbf{\textit{wp}}} (\bigwedge R)) = \bigwedge \{\, (Q \mathbin{\textbf{\textit{wp}}} r) \mid r \in R \,\}$

L8 $(Q \mathbin{\textbf{\textit{wp}}} \mathit{false}) = \mathit{false} \quad \textbf{\textit{if}} \quad Q \,;\, \textbf{\textit{true}} = \textbf{\textit{true}}$

A representative selection of Isabelle proofs for these rules is shown below. Most of them can be proved automatically.

theorem *SemiR-wp*: $(P\ ;\ Q)\ wp\ R = P\ wp\ (Q\ wp\ R)$
by (*utp-rel-auto-tac*)

theorem *CondR-wp*:
assumes
$(P \in WF\text{-}RELATION)$ $(Q \in WF\text{-}RELATION)$
$(b \in WF\text{-}CONDITION)$ $(R \in WF\text{-}RELATION)$
shows ‘$(P \lhd b \rhd Q)\ wp\ R$‘ = ‘$(P\ wp\ R) \lhd b \rhd (Q\ wp\ R)$‘
proof −
have ‘$(P \lhd b \rhd Q)\ wp\ R$‘ = ‘$\neg\ ((P \lhd b \rhd Q)\ ;\ (\neg\ R))$‘
by (*simp add*: *WeakPrecondP-def*)
also from *assms* **have** ... = ‘$\neg\ ((P\ ;\ (\neg\ R)) \lhd b \rhd (Q\ ;\ (\neg\ R)))$‘
by (*simp add*:*CondR-SemiR-distr closure*)
also have ... = ‘$(P\ wp\ R) \lhd b \rhd (Q\ wp\ R)$‘
by (*utp-pred-auto-tac*)
finally show *?thesis* **.**
qed

theorem *ChoiceP-wp*:
$(P \sqcap Q)\ wp\ R$ = ‘$(P\ wp\ R) \wedge (Q\ wp\ R)$‘
by (*utp-rel-auto-tac*)

theorem *ImpliesP-precond-wp*:
‘$[R \Rightarrow S]$‘ $\Longrightarrow$ ‘$[(Q\ wp\ R) \Rightarrow (Q\ wp\ S)]$‘
by (*metis ConjP-wp RefP-AndP RefP-def less-eq-WF-PREDICATE-def*)

theorem *FalseP-wp*:
$Q\ ;\ true = true \Longrightarrow Q\ wp\ false = false$
by (*simp add*:*WeakPrecondP-def*)

9 Designs

The problem pointed out in Section 6—that the relational model does not capture the semantics of nonterminating programs—can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\textbf{\textit{true}}\ ;\ P\ =\ \textbf{\textit{true}}$$

In particular, in Example 13 we presented a non-terminating loop which, when followed by an assignment, behaves like the assignment. Operationally, it is as though the non-terminating loop could be ignored.

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called *ok*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, we observe that *false* cannot possibly belong to this set, since $\textbf{\textit{true}}\ ;\ \textbf{\textit{false}} = \textbf{\textit{false}}$.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements

used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [16], Z [33], and refinement calculi [17,2,18].

In designs, ok records that the program has started, and ok' records that it has terminated. These are auxiliary variables, in the sense that they appear in a design's alphabet, but they never appear in code or in preconditions and postconditions.

In implementing a design, we are allowed to assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that the program terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q, for predicates P and Q not containing ok or ok', is written $(\, P \vdash Q \,)$. It is defined as follows.

$$(\, P \vdash Q \,) \;\widehat{=}\; (\, ok \wedge P \Rightarrow ok' \wedge Q \,)$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Example 14 (Specifications). Suppose that we have two program variables, x and y, and that we want to specify an operation on the state that reduces the value of x but keeps y constant. Furthermore, suppose that x and y take on values that are natural numbers. We could specify this operation in Z using an operation schema[28,33]:

Dec

$x, y, x', y' : \mathbb{N}$

$x > 0$
$x' < x$
$y' = y$

This specifies the decrement operation *Dec* involving the state before the operation (x and y) and the state after the operation x' and y'). The value of x must be strictly positive, or else the invariant that x is always a natural number cannot be satisfied. The after-value x' must be strictly less than the before-value x and the value of y is left unchanged. This Z schema defines its operation as a single relation, just like the alphabetised relations already introduced.

The refinement calculus [17] is similar to Z, except that the relation specifying a program is slit into a precondition and a postcondition, with the meaning described above: if the program is activated in a state satisfying the precondition, then the program must terminate and when it does, the postcondition will be true. Our operation is specified like this:

$$Dec \;\widehat{=}\; x : [\, x > 0, x < x_0 \,]$$

Here, the before-value of x in the postcondition is specified as x_0 and the after-value as simply x. The frame, written before the precondition-postcondition

pair, specifies that only the variable x may be changed; y must remain constant. Something similar happens in VDM [16]:

```
operation Dec()
ext wr x: Nat
pre    x > 0
post   x < x~
```

In UTP, the same operation is specified using a design; the frame is specified by saying what must remain constant:

$$Dec \mathrel{\widehat{=}} (\, x > 0 \vdash x' < x \,)_{+y}$$

□

9.1 Lattice Operators

Abort and miracle are defined as designs in the following examples. Abort has precondition ***false*** and is never guaranteed to terminate. It is denoted by $\bot_{\mathbf{D}}$.

Example 15 (Abort).

$$\begin{array}{lr}
\quad \textbf{\textit{false}} \vdash \textbf{\textit{false}} & \textit{definition of design} \\
= ok \wedge \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textbf{\textit{false}}\ \textit{zero for conjunction} \\
= \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textit{vacuous implication} \\
= \textbf{\textit{true}} & \textit{vacuous implication} \\
= \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{true}} & \textbf{\textit{false}}\ \textit{zero for conjunction} \\
= ok \wedge \textbf{\textit{false}} \Rightarrow ok' \wedge \textbf{\textit{true}} & \textit{definition of design} \\
= \textbf{\textit{false}} \vdash \textbf{\textit{true}} & \square
\end{array}$$

□

Miracle has precondition ***true***, and establishes the impossible: ***false***. It is denoted by $\top_{\mathbf{D}}$.

Example 16 (Miracle).

$$\begin{array}{lr}
\quad \textbf{\textit{true}} \vdash \textbf{\textit{false}} & \textit{definition of design} \\
= ok \wedge \textbf{\textit{true}} \Rightarrow ok' \wedge \textbf{\textit{false}} & \textbf{\textit{true}}\ \textit{unit for conjunction} \\
= ok \Rightarrow \textbf{\textit{false}} & \textit{contradiction} \\
= \neg\, ok & \square
\end{array}$$

□

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\top_{\mathbf{D}} \mathrel{\widehat{=}} (\, \textbf{\textit{true}} \vdash \textbf{\textit{false}} \,) = \neg\, ok$$
$$\bot_{\mathbf{D}} \mathrel{\widehat{=}} (\, \textbf{\textit{false}} \vdash \textbf{\textit{true}} \,) = \textbf{\textit{true}}$$

The least upper bound and the greatest lower bound are established in the following theorem.

Theorem 1. *Meets and joins*

$$\bigsqcap_i (\, P_i \vdash Q_i \,) = (\, \bigwedge_i P_i \,) \vdash (\, \bigvee_i Q_i \,)$$

$$\bigsqcup_i (\, P_i \vdash Q_i \,) = (\, \bigvee_i P_i \,) \vdash (\, \bigwedge_i P_i \Rightarrow Q_i \,)$$

As with the binary choice, the choice $\bigsqcap_i (\, P_i \vdash Q_i \,)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs does, and the postcondition established is that of any of the terminating designs.

Example 17 (Not a design). Notice that designs are not closed under negation.

$$\begin{array}{ll} & \neg\, (P \vdash Q) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{design} \\ = & \neg\, (ok \wedge p \Rightarrow ok' \wedge Q) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{propositional calculus} \\ = & ok \wedge p \wedge (ok' \Rightarrow \neg\, Q) \end{array}$$

Although the negation of a design is not itself a design, this derivation does give a useful identity. □

9.2 Refinement of Designs

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 91 (Refinement of designs)

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \;=\; [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] \qquad \square$$

Proof

$$\begin{array}{ll} & P_1 \vdash Q_1 \;\sqsubseteq\; P_2 \vdash Q_2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{definition of } \sqsubseteq \\ = & [\, (\, P_2 \vdash Q_2 \,) \Rightarrow (\, P_1 \vdash Q_1 \,) \,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{definition of design, twice} \\ = & [\, (\, ok \wedge P_2 \Rightarrow ok' \wedge Q_2 \,) \Rightarrow (\, ok \wedge P_1 \Rightarrow ok' \wedge Q_1 \,) \,] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{case analysis on } ok \\ = & [\, (\, P_2 \Rightarrow ok' \wedge Q_2 \,) \Rightarrow (\, P_1 \Rightarrow ok' \wedge Q_1 \,) \,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{case analysis on } ok' \\ = & [\, (\, (\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,) \,) \wedge (\, \neg\, P_2 \Rightarrow \neg\, P_1 \,) \,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{propositional calculus} \\ = & [\, (\, (\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,) \,) \wedge (\, P_1 \Rightarrow P_2 \,) \,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{predicate calculus} \\ = & [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square \end{array}$$

9.3 Nontermination

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

L1 $\quad \textbf{\textit{true}} \,;\, (P \vdash Q) = \textbf{\textit{true}}$ *left-zero*

Proof

$$
\begin{array}{ll}
& \textbf{\textit{true}} \,;\, (P \vdash Q) & \textit{property of sequential composition} \\
= & \exists\, ok_0 \bullet \textbf{\textit{true}} \,;\, (P \vdash Q)[ok_0/ok] & \textit{case analysis} \\
= & (\, \textbf{\textit{true}} \,;\, (P \vdash Q)[true/ok] \,) \lor (\, \textbf{\textit{true}} \,;\, (P \vdash Q)[false/ok] \,) & \textit{property of design} \\
= & (\, \textbf{\textit{true}} \,;\, (P \vdash Q)[true/ok] \,) \lor (\, \textbf{\textit{true}} \,;\, \textbf{\textit{true}} \,) & \textit{relational calculus} \\
= & (\, \textbf{\textit{true}} \,;\, (P \vdash Q)[true/ok] \,) \lor \textbf{\textit{true}} & \textit{propositional calculus} \\
= & \textbf{\textit{true}} & \square
\end{array}
$$

9.4 Assignment

In this new setting, it is necessary to redefine assignment and skip, as those introduced previously are not designs.

$$(x := e) \;\widehat{=}\; (\, \textbf{\textit{true}} \vdash x' = e \land y' = y \land \cdots \land z' = z \,)$$

$$\mathit{II}_{\mathbf{D}} \;\widehat{=}\; (\, \textbf{\textit{true}} \vdash \mathit{II} \,)$$

Their existing laws hold, but it is necessary to prove them again, as their definitions changed.

L2 $\quad (v := e \,;\, v := f(v)) = (v := f(e))$

L3 $\quad (v := e \,;\, (P \lhd b(v) \rhd Q)) = ((v := e \,;\, P) \lhd b(e) \rhd (v := e \,;\, Q))$

L4 $\quad (\mathit{II}_{\mathbf{D}} \,;\, (P \vdash Q)) = (P \vdash Q)$

As as an example, we present the proof of **L2**.

Proof of **L2**

$$
\begin{array}{ll}
& v := e \,;\, v := f(v) & \textit{definition of assignment, twice} \\
= & (\, \textbf{\textit{true}} \vdash v' = e \,) \,;\, (\, \textbf{\textit{true}} \vdash v' = f(v) \,) & \textit{case analysis on } ok_0 \\
= & (\, (\, \textbf{\textit{true}} \vdash v' = e \,)[true/ok'] \,;\, (\, \textbf{\textit{true}} \vdash v' = f(v) \,)[true/ok] \,) \lor & \\
& \neg\, ok \,;\, \textbf{\textit{true}} & \textit{definition of design} \\
= & (\, (\, ok \Rightarrow v' = e \,) \,;\, (\, ok' \land v' = f(v) \,) \,) \lor \neg\, ok & \textit{relational calculus} \\
= & ok \Rightarrow (\, v' = e \,;\, (\, ok' \land v' = f(v) \,) \,) & \textit{assignment composition} \\
= & ok \Rightarrow ok' \land v' = f(e) & \textit{definition of design} \\
= & (\, \textbf{\textit{true}} \vdash v' = f(e) \,) & \textit{definition of assignment} \\
= & v := f(e) & \square
\end{array}
$$

9.5 Closure under the Program Combinators

If any of the program operators are applied to designs, then the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. The choice between two designs is guaranteed to terminate when they both terminate; since either of them may be chosen, then either postcondition may be established.

T1 $$((P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2)) = (P_1 \wedge P_2 \vdash Q_1 \vee Q_2)$$

If the choice between two designs depends on a condition b, then so do the precondition and the postcondition of the resulting design.

T2 $$((P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2)) = ((P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2))$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and Q_1 is guaranteed to establish P_2. On termination, the sequence establishes the composition of the postconditions.

T3 $$((P_1 \vdash Q_1) ; (P_2 \vdash Q_2)) = ((\neg (\neg P_1 ; \mathbf{true}) \wedge (Q_1 \mathbf{wp} P_2)) \vdash (Q_1 ; Q_2))$$

where $Q_1 \mathbf{wp} P_2$ is the weakest precondition under which execution of Q_1 is guaranteed to achieve the postcondition P_2. It is defined in [15] as

$$Q \mathbf{wp} P = \neg (Q ; \neg P)$$

The Isabelle proof of this fact is difficult, but rewarding:

Example 18 (Isar Proof of Design Composition).

```
theorem DesignD-composition:
assumes
  (P1 ∈ WF-RELATION) (P2 ∈ WF-RELATION)
  (Q1 ∈ WF-RELATION) (Q2 ∈ WF-RELATION)
shows ‘(P1 ⊢ Q1); (P2 ⊢ Q2)‘=‘(¬(¬P1 ; true)) ∧ (Q1 wp P2) ⊢ (Q1; Q2)‘
proof −
  have ‘(P1 ⊢ Q1) ; (P2 ⊢ Q2)‘
       = ‘∃ okay´´´ . ((P1 ⊢ Q1)[$okay´´´/okay´] ; (P2 ⊢ Q2)[$okay´´´/okay])‘
    by (smt DesignD-rel-closure MkPlain-UNDASHED SemiR-extract-variable assms)

  also have ... = ‘ ((P1 ⊢ Q1)[false/okay´] ; (P2 ⊢ Q2)[false/okay])
                  ∨ ((P1 ⊢ Q1)[true/okay´] ; (P2 ⊢ Q2)[true/okay])‘
    by (simp add:ucases typing usubst defined closure unrest DesignD-def assms)

  also have ... = ‘((ok ∧ P1 ⇒ Q1) ; (P2 ⇒ ok′ ∧ Q2)) ∨ ((¬ (ok ∧ P1)) ; true)‘
    by (simp add: typing usubst defined unrest DesignD-def OrP-comm assms)
```

also have ... = ‘((¬ (*ok* ∧ *P1*) ; (*P2* ⇒ *ok′* ∧ *Q2*)) ∨ ¬ (*ok* ∧ *P1*) ; *true*)
∨ *Q1* ; (*P2* ⇒ *ok′* ∧ *Q2*)‘
by (*smt OrP-assoc OrP-comm SemiR-OrP-distr ImpliesP-def*)

also have ... = ‘(¬ (*ok* ∧ *P1*) ; *true*) ∨ *Q1* ; (*P2* ⇒ *ok′* ∧ *Q2*)‘
by (*smt SemiR-OrP-distl utp-pred-simps*(*9*))

also have ... = ‘(¬*ok* ; *true*) ∨ (¬*P1* ; *true*) ∨ (*Q1* ; ¬*P2*) ∨ (*ok′* ∧ (*Q1* ; *Q2*))‘
proof −
from *assms* **have** ‘*Q1* ; (*P2* ⇒ *ok′* ∧ *Q2*)‘ = ‘(*Q1* ; ¬*P2*) ∨ (*ok′* ∧ (*Q1* ; *Q2*))‘
by (*smt AndP-comm SemiR-AndP-right-postcond ImpliesP-def SemiR-OrP-distl*)

thus *?thesis* **by** (*smt OrP-assoc SemiR-OrP-distr demorgan2*)
qed

also have ... = ‘(¬ (¬ *P1* ; *true*) ∧ ¬ (*Q1* ; ¬ *P2*)) ⊢ (*Q1* ; *Q2*)‘
proof −
have ‘(¬ *ok*) ; *true* ∨ (¬ *P1*) ; *true*‘ = ‘¬ *ok* ∨ (¬ *P1*) ; *true*‘
by (*simp add*: *SemiR-TrueP-precond closure*)

thus *?thesis*
by (*smt DesignD-def ImpliesP-def OrP-assoc demorgan2 demorgan3*)
qed

finally show *?thesis* **by** (*simp add*:*WeakPrecondP-def*)
qed

Preconditions can be relations, and this fact complicates the statement of Law ***T3***; if the P_1 is a condition instead, then the law is simplified as follows.

T3′ $(\,(p_1 \vdash Q_1)\,;(P_2 \vdash Q_2)\,) = (\,p_1 \wedge (Q_1 \;\mathbf{wp}\; P_2)) \vdash (Q_1\,;\,Q_2)\,)$

Example 19 (Simplifying condition-composition).

$$
\begin{array}{lr}
\quad \neg\,(\neg\, p_1\,;\, \mathbf{true}) & \textit{composition} \\
= \neg\, \exists\, v_0 \bullet \neg\, p_1[v_0/v'] \wedge \mathbf{true}[v_0/v] & v \textit{ not free in } \mathbf{true} \\
= \neg\, \exists\, v_0 \bullet \neg\, p_1[v_0/v'] \wedge \mathbf{true} & \textit{unit for conjunction} \\
= \neg\, \exists\, v_0 \bullet \neg\, p_1[v_0/v'] & v' \textit{ not free in } p_1 \\
= \neg\, \exists\, v_0 \bullet \neg\, p_1 & v_0 \textit{ not free in } p_1 \\
= \neg\, \neg\, p_1 & \textit{propositional calculus} \\
= p_1 &
\end{array}
$$

□

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

T4 $(\mu\, X, Y \bullet (\, F(X, Y) \vdash G(X, Y)\,)) = (\, P(Q) \vdash Q\,)$

where $P(Y) = (\,\nu X \bullet F(X, Y)\,)$ ***and*** $Q = (\,\mu\, Y \bullet P(Y) \Rightarrow G(P(Y), Y)\,)$

Further intuition comes from the realisation that we want the least refined fixed-point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

10 Healthiness Conditions

Another way of characterising the set of designs is by imposing healthiness conditions on the alphabetised predicates. Hoare & He [15] identify four healthiness conditions that they consider of interest: ***H1*** to ***H4***. We discuss each of them.

10.1 *H1*: Unpredictability

A relation R is ***H1*** healthy if and only if $R = (ok \Rightarrow R)$. This means that observations cannot be made before the program has started. This healthiness condition is idempotent.

Law 101 (*H1* idempotent)

$$\textbf{\textit{H1}} \circ \textbf{\textit{H1}} = \textbf{\textit{H1}}$$

Proof:

$$\begin{array}{lr}
\quad \textbf{\textit{H1}} \circ \textbf{\textit{H1}}(P) & \textbf{\textit{H1}} \\
= ok \Rightarrow (ok \Rightarrow P) & \textit{propositional calculus} \\
= ok \wedge ok \Rightarrow P & \textit{propositional calculus} \\
= ok \Rightarrow P & \textbf{\textit{H1}} \\
= \textbf{\textit{H1}}(P) & \Box
\end{array}$$

*Example 20 (Examples of **H1** relations).*

1. Abort, the bottom of the lattice, is healthy.

$$\mathbf{H1}(\mathbf{true}) \;=\; (ok \Rightarrow \mathbf{true}) \;=\; \mathbf{true}$$

2. Miracle, the top of the lattice, is healthy.

$$\mathbf{H1}(\neg\, ok) \;=\; (ok \Rightarrow \neg\, ok) \;=\; \neg\, ok$$

3. The following relation is healthy: $(ok \wedge x \neq 0 \Rightarrow x' < x)$.

$$\begin{array}{l}\mathbf{H1}(ok \wedge x \neq 0 \Rightarrow x' < x)\\ = ok \Rightarrow (ok \wedge x \neq 0 \Rightarrow x' < x)\\ = (ok \wedge x \neq 0 \Rightarrow x' < x)\end{array}$$

4. The following design is healthy: $(x \neq 0 \vdash x' < x)$.

$$\mathbf{H1}(x \neq 0 \vdash x' < x) \;=\; ok \Rightarrow (x \neq 0 \vdash x' < x) \;=\; (x \neq 0 \vdash x' < x)$$

□

If R is **H1**-healthy, then R also satisfies the left-zero and unit laws below.

$$\mathbf{true} \,;\, R = \mathbf{true} \quad \text{and} \quad \mathbb{I}_D \,;\, R = R$$

We now present a proof of these results. First, we prove that the algebraic unit and zero properties guarantee **H1**-healthiness.

*Designs with Left-Units and Left-Zeros Are **H1**.*

$$\begin{array}{lr}
R & \textit{assumption } (\mathbb{I}_D \textit{ is left-unit}) \\
= \mathbb{I}_D \,;\, R & \mathbb{I}_D \textit{ definition} \\
= (\, \mathbf{true} \vdash \mathbb{I}_D \,) \,;\, R & \textit{design definition} \\
= (\, ok \Rightarrow ok' \wedge \mathbb{I} \,) \,;\, R & \textit{relational calculus} \\
= (\, \neg\, ok \,;\, R \,) \vee (\, \mathbb{I} \,;\, R \,) & \textit{relational calculus} \\
= (\, \neg\, ok \,;\, \mathbf{true} \,;\, R \,) \vee (\, \mathbb{I} \,;\, R \,) & \textit{assumption } (\mathbf{true} \textit{ is left-zero}) \\
= \neg\, ok \vee (\, \mathbb{I} \,;\, R \,) & \textit{assumption } (\mathbb{I} \textit{ is left-unit}) \\
= \neg\, ok \vee R & \textit{relational calculus} \\
= ok \Rightarrow R & \square
\end{array}$$

The Isabelle proof has a few more steps, but follows a similar line of reasoning. We require that P be a well-formed relation, consisting of only undashed and dashed variables. We also prefer the use of the simplifier, executed by simp, to discharge each of the steps.

```
theorem H1-algebraic-intro:
  assumes
    R ∈ WF-RELATION
    (true ; R = true)
    (II_D ; R = R)
  shows R is H1
proof −
  let ?vs = REL-VAR − {okay,okay´}
  have R = II_D ; R by (simp add: assms)
  also have ... = ‘(true ⊢ II_?vs) ; R‘
    by (simp add:SkipD-def)
  also have ... = ‘(ok ⇒ (ok′ ∧ II_?vs)) ; R‘
    by (simp add:DesignD-def)
  also have ... = ‘(ok ⇒ (ok ∧ ok′ ∧ II_?vs)) ; R‘
    by (smt ImpliesP-export)
  also have ... = ‘(ok ⇒ (ok ∧ $okay´ = $okay ∧ II_?vs)) ; R‘
    by (simp add:VarP-EqualP-aux typing defined, utp-rel-auto-tac)
  also have ... = ‘(ok ⇒ II) ; R‘
    by (simp add:SkipRA-unfold[THEN sym]
        SkipR-as-SkipRA ImpliesP-export[THEN sym])
  also have ... = ‘((¬ ok) ; R ∨ R)‘
    by (simp add:ImpliesP-def SemiR-OrP-distr)
  also have ... = ‘(((¬ ok) ; true) ; R ∨ R)‘
    by (simp add:SemiR-TrueP-precond closure)
  also have ... = ‘((¬ ok) ; true ∨ R)‘
    by (simp add:SemiR-assoc[THEN sym] assms)
  also have ... = ‘ok ⇒ R‘
    by (simp add:SemiR-TrueP-precond closure ImpliesP-def)
  finally show ?thesis by (simp add:is-healthy-def H1-def)
qed
```

Next, we prove the implication the other way around: that ***H1***-healthy predicates have the unit and zero properties.

H1 *Predicates Have a Left-Zero.*

true ; R	*assumption* (R *is* ***H1***)
= ***true*** ; ($ok \Rightarrow R$)	*relational calculus*
= (***true*** ; $\neg\ ok$) ∨ (***true*** ; R)	*relational calculus*
= ***true*** ∨ (***true*** ; R)	*relational calculus*
= ***true***	□

. . . and the same in Isabelle:

theorem *H1-left-zero*:
assumes
$P \in$ *WF-RELATION*
P is H1
shows *true* ; P = *true*
proof −
from *assms* **have** ‘*true* ; P‘ = ‘*true* ; $(ok \Rightarrow P)$‘
by (*simp add*:*is-healthy-def H1-def*)
also have ... = ‘*true* ; $(\neg\ ok \vee P)$‘
by (*simp add*:*ImpliesP-def*)
also have ... = ‘$(true\ ;\ \neg\ ok) \vee (true\ ;\ P)$‘
by (*simp add*:*SemiR-OrP-distl*)
also from *assms* **have** ... = ‘*true* $\vee$ $(true\ ;\ P)$‘
by (*simp add*:*SemiR-precond-left-zero closure*)
finally show *?thesis* **by** *simp*
qed

H1 *Predicates Have a Left-Unit.*

$$
\begin{array}{lr}
\mathbb{I}_D\ ;\ R & \textit{definition of } \mathbb{I}_D \\
= (\ \textbf{\textit{true}} \vdash \mathbb{I}_D\ )\ ;\ R & \textit{definition of design} \\
= (\ ok \Rightarrow ok' \wedge \mathbb{I}\ )\ ;\ R & \textit{relational calculus} \\
= (\ \neg\ ok\ ;\ R\ ) \vee (\ ok \wedge R\ ) & \textit{relational calculus} \\
= (\ \neg\ ok\ ;\ \textbf{\textit{true}}\ ;\ R\ ) \vee (\ ok \wedge R\ ) & \textbf{\textit{true}}\ \textit{is left-zero} \\
= (\ \neg\ ok\ ;\ \textbf{\textit{true}}\ ) \vee (\ ok \wedge R\ ) & \textit{relational calculus} \\
= \neg\ ok \vee (\ ok \wedge R\ ) & \textit{relational calculus} \\
= ok \Rightarrow R & R\ \textit{is}\ \textbf{\textit{H1}} \\
= R & \square
\end{array}
$$

... and the same in Isabelle:

theorem *H1-left-unit*:
assumes
$P \in$ *WF-RELATION*
P is H1
shows II_D ; $P = P$
proof −
let *?vs* = *REL-VAR* − {*okay*,*okay´*}
have II_D ; P = ‘$(true \vdash II\ _{?vs})$; P‘ **by** (*simp add*:*SkipD-def*)
also have ... = ‘$(ok \Rightarrow ok' \wedge II\ _{?vs})$; P‘
by (*simp add*:*DesignD-def*)
also have ... = ‘$(ok \Rightarrow ok \wedge ok' \wedge II\ _{?vs})$; P‘
by (*smt ImpliesP-export*)
also have ... = ‘$(ok \Rightarrow ok \wedge \$okay' = \$okay \wedge II\ _{?vs})$; P‘
by (*simp add*:*VarP-EqualP-aux*, *utp-rel-auto-tac*)
also have ... = ‘$(ok \Rightarrow II)$; P‘
by (*simp add*: *SkipRA-unfold*[*of okay*] *ImpliesP-export*[*THEN sym*])
also have ... = ‘$((\neg\ ok)\ ;\ P \vee P)$‘

```
    by (simp add:ImpliesP-def SemiR-OrP-distr)
  also have ... = `(((¬ ok) ; true) ; P ∨ P)`
    by (metis NotP-cond-closure SemiR-TrueP-precond VarP-cond-closure)
  also have ... = `((¬ ok) ; (true ; P) ∨ P)`
    by (metis SemiR-assoc)
  also from assms have ... = `(ok ⇒ P)`
    by (simp add:H1-left-zero ImpliesP-def SemiR-TrueP-precond closure)
  finally show ?thesis using assms
    by (simp add:H1-def is-healthy-def)
qed
```

This means that we can use the left-zero and unit laws to exactly characterise ***H1*** healthiness. We can assert this equivalence property in Isabelle by combining the three theorems:

```
theorem H1-algebraic:
  assumes R ∈ WF-RELATION
  shows R is H1 ⟷ (true ; R = true) ∧ (II_D ; R = R)
   by (metis H1-algebraic-intro H1-left-unit H1-left-zero assms)
```

The design identity is the obvious lifting of the relational identity to a design; that is, it has precondition ***true*** and the postcondition is the relational identity. There's a simple relationship between them: ***H1***.

Law 102 (Relational and design identities)

$$\mathit{II}_D \;=\; \textbf{\textit{H1}}(\mathit{II})$$

Proof:

$$
\begin{array}{lr}
\quad \mathit{II}_D & \mathit{II}_D \\
= (\textbf{\textit{true}} \vdash \mathit{II}) & \textit{design} \\
= (ok \Rightarrow ok' \wedge \mathit{II}) & \mathit{II},\ \textit{prop calculus} \\
= (ok \Rightarrow \mathit{II}) & \textbf{\textit{H1}} \\
= \textbf{\textit{H1}}(\mathit{II}) & \square
\end{array}
$$

```
theorem H1-algebraic-intro:
  assumes
    R ∈ WF-RELATION
    (true ; R = true)
    (II_D ; R = R)
  shows R is H1
proof −
  let ?vs = REL-VAR − {okay,okay´}
  have R = II_D ; R by (simp add: assms)
  also have ... = `(true ⊢ II_?vs) ; R`
    by (simp add:SkipD-def)
  also have ... = `(ok ⇒ (ok′ ∧ II_?vs)) ; R`
    by (simp add:DesignD-def)
  also have ... = `(ok ⇒ (ok ∧ ok′ ∧ II_?vs)) ; R`
```

by (*smt ImpliesP-export*)
also have ... = ‘(*ok* ⇒ (*ok* ∧ $*okay*´ = $*okay* ∧ *II* $_{?vs}$)) ; *R*‘
by (*simp add*: *VarP-EqualP-aux typing defined*, *utp-rel-auto-tac*)
also have ... = ‘(*ok* ⇒ *II*) ; *R*‘
by (*simp add*:*SkipRA-unfold*[*THEN sym*]
SkipR-as-SkipRA ImpliesP-export[*THEN sym*])
also have ... = ‘((¬ *ok*) ; *R* ∨ *R*)‘
by (*simp add*:*ImpliesP-def SemiR-OrP-distr*)
also have ... = ‘(((¬ *ok*) ; *true*) ; *R* ∨ *R*)‘
by (*simp add*:*SemiR-TrueP-precond closure*)
also have ... = ‘((¬ *ok*) ; *true* ∨ *R*)‘
by (*simp add*:*SemiR-assoc*[*THEN sym*] *assms*)
also have ... = ‘*ok* ⇒ *R*‘
by (*simp add*:*SemiR-TrueP-precond closure ImpliesP-def*)
finally show *?thesis* **by** (*simp add*:*is-healthy-def H1-def*)
qed

10.2 *H2*: Possible Termination

The second healthiness condition is $[\, R[\mathit{false}/ok'] \Rightarrow R[\mathit{true}/ok']\,]$. This means that if R is satisfied when ok' is *false*, it is also satisfied then ok' is *true*. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

Example 21 (Example ***H2*** *predicates).*

1. $\bot_{\boldsymbol{D}}$

$$\bot_{\boldsymbol{D}}^{f} \;=\; \textbf{\textit{true}}^{f} \;=\; \textbf{\textit{true}} \;=\; \textbf{\textit{true}}^{t} \;=\; \bot_{\boldsymbol{D}}^{t}$$

2. $\top_{\boldsymbol{D}}$

$$\top^{f} \;=\; (\neg\, ok)^{f} \;=\; \neg\, ok \;=\; (\neg\, ok)^{t} \;=\; \top_{\boldsymbol{D}}^{t}$$

3. $(ok' \wedge (x' = 0))$

$$(ok' \wedge (x' = 0))^{f} \;=\; \textbf{\textit{false}} \;\Rightarrow\; (x' = 0) \;=\; (ok' \wedge x' = 0)^{t}$$

4. $(x \neq 0 \vdash x' < x)$

$$\begin{aligned}
&(x \neq 0 \vdash x' < x)^{f} \\
&= (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^{f} \\
&= (ok \wedge x \neq 0 \Rightarrow \textbf{\textit{false}}) \\
&\Rightarrow (ok \wedge x \neq 0 \Rightarrow x' < x) \\
&= (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^{t} \\
&= (x \neq 0 \vdash x' < x)^{t}
\end{aligned}$$

□

The healthiness condition ***H2*** is not obviously characterised by a monotonic idempotent function. We now define the idempotent J for alphabet $\{ok, ok', v, v'\}$, and use this in an alternative definition of ***H2***.

$$J \mathrel{\widehat{=}} (ok \Rightarrow ok') \wedge v' = v$$

The most interesting property of J is the following algebraic law that allows a relation to be split into two complementary parts, one that definitely aborts and one that does not. Note the asymmetry between the two parts.

Law 103 (J-split) *For all relations with ok and ok' in their alphabet,*

$$P ; J = P^f \vee (P^t \wedge ok')$$

Proof:

$$\begin{array}{lr}
P ; J & J \\
= P ; (ok \Rightarrow ok') \wedge v' = v & \textit{propositional calculus} \\
= P ; (ok \Rightarrow ok \wedge ok') \wedge v' = v & \textit{propositional calculus} \\
= P ; (\neg\, ok \vee ok \wedge ok') \wedge v' = v & \textit{relational calculus} \\
= P ; \neg\, ok \wedge v' = v & \textit{right one-point, twice} \\
\quad \vee & \\
\quad (P ; ok \wedge v' = v) \wedge ok' & \\
= P^f \vee (P^t \wedge ok') & \square
\end{array}$$

Likewise this proof can be mechanised in Isabelle, though a little more detailed is required. In particular, we treat the equalities of each sides of the disjunction separately in the final step.

```
theorem J-split:
  assumes P ∈ WF-RELATION
  shows ‘P ; J‘ = ‘P^f ∨ (P^t ∧ ok′)‘
proof −
  let ?vs = (REL-VAR − {okay,okay´})

  have ‘P ; J‘ = ‘P ; ((ok ⇒ ok′) ∧ II ?vs)‘ by (simp add:J-pred-def)

  also have ... = ‘P ; ((ok ⇒ ok ∧ ok′) ∧ II ?vs)‘ by (smt ImpliesP-export)

  also have ... = ‘P ; ((¬ ok ∨ (ok ∧ ok′)) ∧ II ?vs)‘ by (utp-rel-auto-tac)

  also have ... = ‘(P ; (¬ ok ∧ II ?vs)) ∨ (P ; (ok ∧ (II ?vs ∧ ok′)))‘
    by (smt AndP-OrP-distr AndP-assoc AndP-comm SemiR-OrP-distl)

  also have ... = ‘P^f ∨ (P^t ∧ ok′)‘
  proof −
    from assms have ‘(P ; (¬ ok ∧ II ?vs))‘ = ‘P^f ‘
      by (simp add: SemiR-left-one-point SkipRA-right-unit )
```

```
  moreover have ‘(P ; (ok ∧ II ?vs ∧ ok′)) ‘ = ‘(P^t ∧ ok′)‘
  proof −
    from assms have ‘(P ; (ok ∧ II ?vs ∧ ok′)) ‘ = ‘(P ; (ok ∧ II ?vs)) ∧ ok′‘
      by (utp-xrel-auto-tac)

    moreover from assms have ‘(P ; (ok ∧ II ?vs)) ‘ = ‘P^t ‘
      by (simp add: SemiR-left-one-point SkipRA-right-unit )

    finally show ?thesis .
  qed

  ultimately show ?thesis by simp
 qed

 finally show ?thesis .
qed
```

The two characterisations of ***H2*** are equivalent.

Law 104 (*H2* equivalence)

$$(P = P\ ;\ J)\ =\ [\,P^f \Rightarrow P^t\,]$$

Proof:

$$\begin{aligned}
&(P = P\ ;\ J) && \textit{J-split}\\
&= (P = P^f \vee (P^t \wedge ok')) && \textit{ok' split}\\
&= (P = P^f \vee (P^t \wedge ok'))^f \wedge (P = P^f \vee (P^t \wedge ok'))^t && \textit{subst.}\\
&= (P^f = P^f \vee (P^t \wedge \textbf{\textit{false}})) \wedge (P^t = P^f \vee (P^t \wedge \textbf{\textit{true}})) && \textit{prop calc.}\\
&= (P^f = P^f) \wedge (P^t = P^f \vee P^t) && \textit{reflection}\\
&= (P^t = P^f \vee P^t) && \textit{predicate calculus}\\
&= [\,P^f \Rightarrow P^t\,] && \square
\end{aligned}$$

. . . and in Isabelle:

```
 theorem H2-equivalence:
   assumes R ∈ WF-RELATION
   shows R is H2 ⟷ [R^f ⇒ R^t]
 proof −
   from assms have ‘[R ⇔ (R ; J)]‘ = ‘[R ⇔ (R^f ∨ (R^t ∧ ok′))]‘
     by (simp add:J-split)

   also have ... = ‘[(R ⇔ R^f ∨ R^t ∧ ok′)^f ∧ (R ⇔ R^f ∨ R^t ∧ ok′)^t]‘
     by (simp add:ucases)

   also have ... = ‘[(R^f ⇔ R^f) ∧ (R^t ⇔ R^f ∨ R^t)]‘
     by (simp add:usubst closure typing defined)
```

```
  also have ... = ‘[R^t ⇔ (R^f ∨ R^t)]‘
    by (utp-pred-tac)

  finally show ?thesis
    by (utp-pred-auto-tac)
qed
```

J itself is ***H2*** healthy.

Law 105 (J is *H2*)

$$J = \mathbf{H2}(J)$$

Proof:

$$\begin{aligned}
& \mathbf{H2}(J) && J\text{-split} \\
&= J^f \vee (J^t \wedge ok') && J \\
&= (\neg\, ok \wedge v' = v) \vee (ok' \wedge v' = v) && \text{propositional calculus} \\
&= (\neg\, ok \vee ok') \wedge v' = v && \text{propositional calculus} \\
&= (ok \Rightarrow ok') \wedge v' = v && J \\
&= J && \square
\end{aligned}$$

...and in Isabelle:

```
theorem J-is-H2:
  H2(J) = J
proof −
  let ?vs = (REL-VAR − {okay,okay´})
  have H2(J) = ‘J^f ∨ (J^t ∧ ok')‘
    by (metis H2-def J-closure J-split)

  also have ... = ‘((¬ ok ∧ II ?vs) ∨ II ?vs ∧ ok')‘
    by (simp add:J-pred-def usubst typing defined closure)

  also have ... = ‘(¬ ok ∨ ok') ∧ II ?vs‘
    by (utp-pred-auto-tac)

  also have ... = ‘(ok ⇒ ok') ∧ II ?vs‘
    by (utp-pred-tac)

  finally show ?thesis
    by (metis J-pred-def)
qed
```

J is idempotent.

Law 106 (*H2*-idempotent)

$$\mathit{H2} \circ \mathit{H2} \;=\; \mathit{H2}$$

Proof:

$$\begin{array}{lr}
\quad \mathit{H2} \circ \mathit{H2}(P) & \mathit{H2} \\
= (P \,;\, J) \,;\, J & \textit{associativity} \\
= P \,;\, (J \,;\, J) & \mathit{H2} \\
= P \,;\, \mathit{H2}(J) & J \ \mathit{H2}\ \textit{healthy} \\
= P \,;\, J & \mathit{H2} \\
= P & \Box
\end{array}$$

. . . and in Isabelle:

```
theorem H2-idempotent:
  ‘H2 (H2 R)‘ = ‘H2 R‘
proof −
  have ‘H2 (H2 R)‘ = ‘(R ; J) ; J‘
    by (metis H2-def)
  also have ... = ‘R ; (J ; J)‘
    by (metis SemiR-assoc)
  also have ... = ‘R ; H2 J‘
    by (metis H2-def)
  also have ... = ‘R ; J‘
    by (metis J-is-H2)
  also have ... = ‘H2 R‘
    by (metis H2-def)
  finally show ?thesis .
qed
```

Any predicate that insists on proper termination is healthy.

*Example 22 (Example: **H2**-substitution).*

$ok' \wedge (x' = 0)$ is ***H2***

Proof:

$$\begin{array}{l}
\quad (ok' \wedge (x' = 0))^f \Rightarrow (ok' \wedge (x' = 0))^t \\
= (\mathit{false} \wedge (x' = 0) \Rightarrow \mathit{true} \wedge (x' = 0)) \\
= (\mathit{false} \Rightarrow (x' = 0)) \\
= \mathit{true}
\end{array}$$

$\Box$

The proof could equally well be done with the alternative characterisation of ***H2***.

Example 23. Example: ***H2***-J

$ok' \wedge (x' = 0)$ is ***H2***

Proof:

$$
\begin{aligned}
& ok' \wedge (x' = 0)\ ;\ J && \textit{J-splitting} \\
= {} & (ok' \wedge (x' = 0))^f \vee ((ok' \wedge (x' = 0))^t \wedge ok') && \textit{subst.} \\
= {} & (\textbf{\textit{false}} \wedge (x' = 0)) \vee (\textbf{\textit{true}} \wedge (x' = 0) \wedge ok') && \textit{prop. calculus} \\
= {} & \textbf{\textit{false}} \vee ((x' = 0) \wedge ok') && \textit{propositional calculus} \\
= {} & ok' \wedge (x' = 0)
\end{aligned}
$$

□

If a relation is both ***H1*** and ***H2*** healthy, then it is a design. We prove this by showing that the relation can be expressed syntactically as a design.

Law 107 (*H1-H2* relations are designs)

$$
\begin{aligned}
& P && \textit{assumption: } P \textit{ is } \textbf{\textit{H1}} \\
= {} & ok \Rightarrow P && \textit{assumption: } P \textit{ is } \textbf{\textit{H2}} \\
= {} & ok \Rightarrow P\ ;\ J && J\textit{-splitting} \\
= {} & ok \Rightarrow P^f \vee (P^t \wedge ok') && \textit{propositional calculus} \\
= {} & ok \wedge \neg\, P^f \Rightarrow ok' \wedge P^t && \textit{design} \\
= {} & \neg\, P^f \vdash P^t && \square
\end{aligned}
$$

Likewise this proof can be formalised in Isabelle:

```
theorem H1-H2-is-DesignD:
  assumes
    P ∈ WF-RELATION
    P is H1
    P is H2
  shows P = `(¬ P^f) ⊢ P^t`
proof −
  have P = `ok ⇒ P`
    by (metis H1-def assms(2) is-healthy-def)

  also have ... = `ok ⇒ (P ;  J)`
    by (metis H2-def assms(3) is-healthy-def)

  also have ... = `ok ⇒ (P^f ∨ (P^t ∧ ok'))`
    by (metis J-split assms(1))

  also have ... = `ok ∧ (¬ P^f) ⇒ ok' ∧ P^t`
    by (utp-pred-auto-tac)
```

also have ... = ‘$\neg\ P^f \vdash P^t$‘
by (*metis DesignD-def*)

finally show *?thesis* **.**
qed

Designs are obviously **H1**; we now show that they must also be **H2**. These two results complete the proof that **H1** and **H2** together exactly characterise designs.

Law 108 *Designs are* **H2**

$$
\begin{array}{ll}
& (\ P \vdash Q\ )^f & \textit{definition of design} \\
= & (\ ok \wedge P \Rightarrow \textbf{false}\ ) & \textit{propositional calculus} \\
\Rightarrow & (\ ok \wedge P \Rightarrow Q\ ) & \textit{definition of design} \\
= & (\ P \vdash Q\ )^t & \square
\end{array}
$$

Miracle, even though it does not mention ok', is **H2**-healthy.

Example 24 (Miracle is **H2***).*

$$
\begin{array}{ll}
& \neg\ ok & \textit{miracle} \\
= & \textbf{true} \vdash \textbf{false} & \textit{designs are } \textbf{H2} \\
= & \textbf{H2}(\textbf{true} \vdash \textbf{false}) & \textit{miracle} \\
= & \textbf{H2}(\neg\ ok)
\end{array}
$$

□

The final thing to prove is that it does not matter in which order we apply **H1** and **H2**; the key point is that a design requires both properties.

Law 109 (*H1-H2* commute)

$$
\begin{array}{ll}
& \textbf{H1} \circ \textbf{H2}(P) & \textbf{H1, H2} \\
= & ok \Rightarrow P\ ;\ J & \textit{propositional calculus} \\
= & \neg\ ok\ \vee\ P\ ;\ J & \textit{miracle is } \textbf{H2} \\
= & \textbf{H2}(\neg\ ok)\ \vee\ P\ ;\ J & \textbf{H2} \\
= & \neg\ ok\ ;\ J\ \vee\ P\ ;\ J & \textit{relational calculus} \\
= & (\neg\ ok \vee P)\ ;\ J & \textit{propositional calculus} \\
= & (ok \Rightarrow P)\ ;\ J & \textbf{H1, H2} \\
= & \textbf{H2} \circ \textbf{H1}(P) & \square
\end{array}
$$

10.3 *H3*: Dischargeable Assumptions

The healthiness condition **H3** is specified as an algebraic law: $R = R\ ;\ \mathbb{I}_D$. A design satisfies **H3** exactly when its precondition is a condition. This is a very

desirable property, since restrictions imposed on dashed variables in a precondition can never be discharged by previous or successive components. For example, $x' = 2 \vdash \mathit{true}$ is a design that can either terminate and give an arbitrary value to x, or it can give the value 2 to x, in which case it is not required to terminate. This is a rather bizarre behaviour.

A Design Is ***H3 iff*** *Its Assumption Is a Condition.*

$$
\begin{aligned}
& (\,(P \vdash Q) = (\,(P \vdash Q)\,;\, \mathbb{I}_D\,)\,) && \textit{definition of design-skip}\\
=\ & (\,(P \vdash Q) = (\,(P \vdash Q)\,;\, (\,\mathbf{true} \vdash \mathbb{I}_D\,)\,)\,) && \textit{sequence of designs}\\
=\ & (\,(P \vdash Q) = (\,\neg\,(\,\neg\, P\,;\, \mathbf{true}\,) \wedge \neg\,(\,Q\,;\, \neg\, \mathbf{true}\,) \vdash Q\,;\, \mathbb{I}_D\,)\,) && \textit{skip unit}\\
=\ & (\,(P \vdash Q) = (\,\neg\,(\,\neg\, P\,;\, \mathbf{true}\,) \vdash Q\,)\,) && \textit{design equality}\\
=\ & (\,\neg\, P = \neg\, P\,;\, \mathbf{true}\,) && \textit{propositional calculus}\\
=\ & (\,P = P\,;\, \mathbf{true}\,) && \square
\end{aligned}
$$

The final line of this proof states that $P = \exists\, v' \bullet P$, where v' is the output alphabet of P. Thus, none of the after-variables' values are relevant: P is a condition only on the before-variables.

10.4 *H4*: Feasibility

The final healthiness condition is also algebraic: $R\ ;\ \mathbf{true} = \mathbf{true}$. Using the definition of sequence, we can establish that this is equivalent to $\exists\, v' \bullet R$, where v' is the output alphabet of R. In words, this means that for *every* initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet: more concisely, establishing a final state is feasible. The design $\top_D$ is not ***H4*** healthy, since miracles are not feasible.

11 Related Work

Our mechanisation of UTP theories of relations and of designs and our future mechanisation of the theory of reactive processes form the basis for reasoning about a family of modern multi-paradigm modelling languages. This family contains both *Circus* [34,35] and CML [37]: *Circus* combines Z [28] and CSP [14], whilst CML combines VDM [16] and CSP. Both languages are based firmly on the notion of refinement [25] and have a variety of extensions with additional computational paradigms, including real-time [27,31], object orientation [8], synchronicity [5], and process mobility [29,30]. Further information on *Circus* may be found at `www.cs.york.ac.uk/circus`. CML is being developed as part of the European Framework 7 COMPASS project on Comprehensive Modelling for Advanced Systems of Systems (grant Agreement: 287829). See `www.compass-research.eu`.

Our implementation of UTP in Isabelle is a natural extension of the work in Oliveira's PhD thesis [20], which is extended in [22], where UTP is embedded in

ProofPowerZ, an extension of ProofPower/HOL supporting the Z notation, to mechanise the definition of *Circus*.

Feliachi et al. [10] have developed a machine-checked, formal semantics based on a shallow embedding of *Circus* in Isabelle/Circus, a semantic theory of UTP also based on Isabelle/HOL. The definitions of *Circus* are based on those in [21], which are in turn based on those in [35]. Feliachi et al. derive proof rules from this semantics and implement tactic support for proofs of refinement for *Circus* processes involving both data and behavioral aspects. This proof environment supports a syntax for the semantic definitions that is close to textbook presentations of Circus.

Our work differs from that of Feliachi et al. in three principle ways:

1. **Alphabets.** We have a unified type for predicates where alphabets are represented explicitly. Predicates with different alphabets can be composed without the need for type-level coercions, and variables can be easily added and removed.
2. **Expressivity.** Our encoding of predicates is highly flexible, providing support for many different operators and subtheories, of which binary relations is only one. We also can support complex operators on variables, in particular we provide a unified notion of substitution. The user can also supply their own type system for values, meaning we are not necessarily limited to the type-system of HOL.
3. **Meta-theoretic Proofs.** We give a deeper semantics to operators such as sequential composition and the quantifiers, rather than identifying them with operators of HOL, and therefore support proofs about the operators of the UTP operators. This meta-level reasoning allows us to perform soundness proofs about the denotational semantics of our language.

12 Conclusion

We have mechanised UTP, including alphabetised predicates, relations, the operators of imperative programming, and the theory of designs. All the proofs contained in this paper have been mechanised within our library, including those where the proofs have been omitted. Thus far, we have mechanised over 200 laws about the basic operators and over 40 laws about the theory of designs. The UTP library can therefore be used to perform basic static analysis of imperative programs. The proof automation level is high due to the inclusion of our proof tactics and the power of sledgehammer combined with the algebraic laws of the UTP.

There are several directions for future work:

- Mechanise additional theories, for instance CSP and *Circus*, which will give the ability to reason about reactive programs with concurrency.
- Complete the VDM/CML model, which includes implementing all the standard VDM library functions, so we can support verification of VDM specifications.

- Implement *proof obligations* for UTP expressions, so that ***H4*** healthiness of a program can be verified.
- Implement Z-schema types, which allow the specification of complex data structures.
- Implement the complete refinement calculus by mechanising refinement laws in Isabelle/UTP. This will allow the derivation of programs from high-level specifications, supported by mechanised proof.

All of this is leading towards proof support for CML, so that we can prove the validity of complex specifications of systems of systems in the COMPASS tool.

References

1. Abrial, J.-R.: The B-Book: Assigning Progams to Meanings. Cambridge University Press (1996)
2. Back, R.J.R., Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer (1998)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011) §
5. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-Circus. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 75–97. Springer, Heidelberg (2007)
6. Beg, A., Butterfield, A.: Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In: Proceedings of the 8th International Conference on Frontiers of Information Technology, FIT 2010, article 47 (2010)
7. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in Unifying Theories of Programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
8. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software and System Modeling 4(3), 277–296 (2005)
9. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/Circus: A Process Specification and Verification Environment. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 243–260. Springer, Heidelberg (2012)
11. Harwood, W., Cavalcanti, A., Woodcock, J.: A theory of pointers for the UTP. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 141–155. Springer, Heidelberg (2008)
12. Hehner, E.C.R.: Retrospective and prospective for Unifying Theories of Programming. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 1–17. Springer, Heidelberg (2006)
13. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister: High-performance equational deduction. Journal of Automated Reasoning 18(2), 265–270 (1997)

14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
15. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science. Prentice Hall (1998)
16. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International (1986)
17. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
18. Morris, J.M.: A Theoretical Basis for Stepwise Refinement and the Programming Calculus. Science of Computer Programming 9(3), 287–306 (1987)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
20. Oliveira, M.V.M.: Formal derivation of state-rich reactive programs using Circus. PhD Thesis, Department of Computer Science, University of York, Report YCST-2006/02 (2005)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput. 21(1-2), 3–32 (2009)
22. Oliveira, M., Cavalcanti, A., Woodcock, J.: Unifying theories in ProofPower-Z. Formal Aspects of Computing 25(1), 133–158 (2013)
23. Perna, J.I., Woodcock, J.: UTP semantics for Handel-C. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 142–160. Springer, Heidelberg (2010)
24. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. Journal of AI Communications 15(2/3), 91–110 (2002)
25. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in Circus. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 451–470. Springer, Heidelberg (2002)
26. Schultz, S.: E—A braniac theorem prover. Journal of AI Communications 15(2/3), 111–126 (2002)
27. Sherif, A., He, J.: Towards a Time Model for Circus. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002)
28. Michael Spivey, J.: The Z Notation: A Reference Manual, 2nd edn. Series in Computer Science. Prentice Hall International (1992)
29. Tang, X., Woodcock, J.: Towards Mobile Processes in Unifying Theories. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, September 28–30, pp. 44–53. IEEE Computer Society (2004)
30. Tang, X., Woodcock, J.: Travelling Processes. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 381–399. Springer, Heidelberg (2004)
31. Wei, K., Woodcock, J., Cavalcanti, A.: *Circus Time* with Reactive Designs. In: Wolff, B., Gaudel, M.-C., Feliachi, A. (eds.) UTP 2012. LNCS, vol. 7681, pp. 68–87. Springer, Heidelberg (2013)
32. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)
33. Woodcock, J., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
34. Woodcock, J., Cavalcanti, A.: A Concurrent Language for Refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, July 16–17. BCS Workshops in Computing, pp. 16–17 (2001)
35. Woodcock, J., Cavalcanti, A.: The Semantics of *Circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)

36. Woodcock, J., Cavalcanti, A.: A tutorial introduction to Designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004)
37. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: A formal modelling language for Systems of Systems. In: 7th IEEE International Conference on System of Systems Engineering (SoSE), pp. 1–6 (2012)
38. Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 238–257. Springer, Heidelberg (2010)
39. Zhu, H., Yang, F., He, J.: Generating denotational semantics from algebraic semantics for event-driven system-level language. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 286–308. Springer, Heidelberg (2010)